Georg Lausen
Dan Suciu (Eds.)

LNCS 2921

Database Programming Languages

9th International Workshop, DBPL 2003
Potsdam, Germany, September 2003
Revised Papers

Springer

Lecture Notes in Computer Science 2921
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Georg Lausen   Dan Suciu (Eds.)

Database Programming Languages

9th International Workshop, DBPL 2003
Potsdam, Germany, September 6-8, 2003
Revised Papers

Springer

Volume Editors

Georg Lausen
University of Freiburg
Institute for Computer Science
Georg-Köhler-Allee 051, 79110 Freiburg, Germany
E-mail: lausen@informatik.uni-freiburg.de

Dan Suciu
University of Washington
Box 352350, Seattle, WA 98195, USA
E-mail: suciu@cs.washington.edu

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): H.2, H.3, E.2, H.4

ISSN 0302-9743
ISBN 3-540-20896-8 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag is a part of Springer Science+Business Media

springeronline.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10983638 06/3142 5 4 3 2 1 0

Preface

The papers in this volume represent the technical program of the 9th Biennial Workshop on Data Bases and Programming Languages (DBPL 2003), which was held on September 6–8, 2003, in Potsdam, Germany. The workshop meets every two years, and is a well-established forum for ideas that lie at the intersection of database and programming language research. DBPL 2003 continued the tradition of excellence initiated by its predecessors in Roscoff, Finistre (1987), Salishan, Oregon (1989), Nafplion, Argolida (1991), Manhattan, New York (1993), Gubbio, Umbria (1995), Estes Park, Colorado (1997), Kinloch Rannoch, Scotland (1999), and Frascati, Rome (2001).

The program committee selected 14 papers out of 22 submissions, and invited two contributions. The 16 talks were presented over three days, in seven sessions.

In the **invited talk** Jennifer Widom presented the paper *CQL: a Language for Continuous Queries over Streams and Relations*, coauthored by Arvind Arasu and Shivnath Babu. While a lot of research has been done recently on query processing over data streams, CQL is virtually the first proposal of a query language on streams that is a strict extension of SQL. The language is structured around a simple yet powerful idea: it has two distinct data types, relations and streams, with well-defined operators for mapping between them. Window specification expressions, such as sliding windows, map streams to relations, while operators such as "insert stream," "delete stream," and "relation stream" map relations to streams by returning, at each moment in time, the newly inserted tuples, the deleted tuples, or a snapshot of the entire relation. The numerous examples in this paper make a convincing case for the power and usefulness of CQL.

The **invited tutorial** was presented by Georg Gottlob and Christoph Koch on *XPath Query Processing*. They described why existing XPath processors run in exponential time in the size of the XPath expression, and showed simple examples of XPath expressions and XML documents on which several popular XPath implementations took unacceptably long time to execute, or even failed to terminate. They described then a class of algorithms based on dynamic programming for evaluating XPath expressions in PTIME, and illustrated several optimization techniques that can be applied to these algorithms.

The three papers of the session **Static Analysis** contain some surprising results on query languages. The first, *Satisfiability of XPath Expressions*, by Jan Hidders, shows that XPath expressions that include parent/ancestor axes, union, intersection, and difference, may not always be satisfiable, i.e., there are expressions that return the empty result on any XML document. The paper analyzes when an XPath expression is satisfiable, and shows that for various combinations of these operators this problem ranges from NP-complete to PTIME. The second paper, *Containment of Relational Queries with Annotation Propagation*, by Wang-Chiew Tan, studies the containment and equivalence problems of relational queries that carry along annotations from source data. The paper shows that certain relational queries that are equivalent under the traditional semantics, become in-equivalent when one takes into account the annotations they carry. As a consequence, one needs to rethink relational query optimization if the queries are expected to return the annotations in the database. The third paper, *Avoiding Unnecessary Ordering Operations in XPath*, by Jan Hidders and Philippe Michiels, notices that often XQuery optimizers introduce unnecessary sort operators to restore the document order of the elements. The paper describes an elegant algorithm for detecting redundant sort operators, which then an optimizer can remove.

In the session **Transactions** three papers addressed various issues related to the integration of transactional semantics in persistent programming languages. *Consistency of Java Transactions*, by Suad Alagic and Jeremy Logan, proposes a model of Java transactions, based on a subtle interplay of constraints, bounded parametric polymorphism, and orthogonal persistence. The paper *Integrating Database and Programming Languages Constraints*, by Oded Shmueli, Mukund Raghavachari, Vivek Sarkar, Rajesh Bordawekar, and Michael Burke, addresses the problem of data consistency in applications that interact with databases. It describes an architecture that automatically inserts checks in the application that ensure data consistency. The third paper, *A Unifying Semantics for Active Databases Using Non-Markovian Theories of Actions*, by Iluju Kiringa and Ray Reiter, develops a new form of theories for modeling active behavior in databases, such as active databases. Their formalism is based on the situation calculus and non-Markovian control.

The session **Modeling Data and Services** contained three papers focusing on Web services, XML constraints, and persistent objects. In *Modeling Dynamic Web Data*, Philippa Gardner and Sergio Maffeis introduced a calculus for describing the dynamic behavior of Web data. The calculus combines semistructured data with an extension of the π-calculus, and can be used to reason about behavior found, for example, in dynamic Web page programming, applet interaction, and service orchestration. In *Semantics of Objectified XML Constraints*, Suad Alagic and David Briggs developed a model theory for a functional object-oriented data model extended with XML-like types, based on Featherweight Java. Finally, the third paper, *M^2ORM^2: a Model for the Transparent Management of Relationally Persistent Objects*, by Luca Cabibbo and Roberto Procelli, describes a “meet in the middle” approach for mapping object-oriented application objects to relational databases.

The session **Novel Applications of XML and XQuery** contained two papers describing some novel applications of the newly standardized XML query language. In *Using XQuery for Flat-File Based Scientific Datasets* the authors, Xiaogang Li and Gagan Agrawal, showed that XQuery is suitable for scientific applications, which were traditionally solved in FORTRAN. The paper describes an application of XQuery to satellite data processing, and proposes a new class of optimization techniques for XQuery to better support these types of applications. The paper *A Query Algebra for Fragmented XML Stream Data*, by Sujoe Bose, Leonidas Fegaras, David Levine, and Vamsi Chaluvadi, describes an application of XQuery for broadcast XML stream data processing. The authors propose an extension of the XML data model with *holes* and *fillers*, and show how XQuery can be adapted to deal with them.

Finally, three papers formed the session **XML Processing and Validation**. The first, *Updates and Incremental Validation of XML Documents*, by Beatrice Bouchou and Mirian Halfeld Ferrari Alves, describes a practical algorithm for incrementally checking the validity of an XML file against a DTD. Starting from a valid XML document, the algorithm checks the validity of a document obtained by inserting, deleting, or modifying an element (subtree) of the document. In *Attribute Grammars for Scalable Query Processing on XML Streams*, the authors, Christoph Koch and Stefanie Scherzinger, described a simple formalism based on attribute grammars that can be used to specify transformations on XML streams. In the third paper, *A General Framework for Estimating XML Query Cardinality*, Carlo Sartiani addressed one of the most difficult problems in XML query optimization, i.e., cardinality estimation, and described a comprehensive approach that applies to the entire XQuery language.

DBPL 2003 was hosted and sponsored by the Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam. We gratefully acknowledge this valuable support. We also thank Mathias Weske and his team for the perfect organization and all the local arrangements.

November 2003 Georg Lausen, Dan Suciu

Organization

DBPL 2003 was organized by the Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam.

Program Chairs

Georg Lausen (University of Freiburg, Germany)
Dan Suciu (University of Washington, USA)

Program Committee

Karl Aberer (EPFL Lausanne)
Phil Bernstein (Microsoft Research)
Mary Fernandez (AT&T Labs)
Zack Ives (University of Pennsylvania)
Trevor Jim (AT&T Labs)
Georg Lausen (University of Freiburg)
Maurizio Lenzerini (University of Rome)
Wolfgang May (University of Göttingen)
Renee Miller (University of Toronto)
Thomas Schwentick (University of Marburg)
Michael Schwartzbach (BRICS)
Jay Shanmugasundaram (Cornell University)
Dan Suciu (University of Washington)
Val Tannen (University of Pennsylvania)

External Referees

Perikles Andritsos
Aske Simon Christensen
Adina Costea
Anwitaman Datta
Guiseppe De Giacomo
Neil Jones
Anastasios Kementsietsidis
Jayant Madhavan
Anders Muller
Antal Novak
Shankar Pal
Feng Shao
Martin Strecker
Yannis Velegrakis

Local Organization

Mathias Weske (University of Potsdam, Germany)

Sponsoring Institution

Hasso Plattner Institute (Potsdam)

Table of Contents

Invited Contributions

Static Analysis

Transactions

Modeling Data and Services

Novel Applications of XML and XQuery

XML Processing and Validation

CQL: A Language for Continuous Queries over Streams and Relations[*]

Arvind Arasu, Shivnath Babu, and Jennifer Widom

Stanford University
{arvinda,shivnath,widom}@cs.stanford.edu

Abstract. Despite the recent surge of research in query processing over data streams, little attention has been devoted to defining precise semantics for continuous queries over streams. We first present an abstract semantics based on several building blocks: formal definitions for streams and relations, mappings among them, and any relational query language. From these basics we define a precise interpretation for continuous queries over streams and relations. We then propose a concrete language, *CQL* (for *Continuous Query Language*), which instantiates the abstract semantics using SQL as the relational query language and window specifications derived from SQL-99 to map from streams to relations. We have implemented most of the CQL language in a Data Stream Management System at Stanford, and we have developed a public repository of data stream applications that includes a wide variety of queries expressed in CQL.

1 Introduction

There has been a considerable surge of interest recently in query processing over *unbounded data streams* [1, 2]. Because of the continuously arriving nature of the data, queries over data streams tend to be *continuous* [3–6], rather than the traditional *one-time* queries over stored data sets. Many papers have included example continuous queries over data streams, expressed in some declarative language, e.g., [7, 8, 4, 5]. However, these queries tend to be for illustrative purposes, and for complex queries the precise semantics may be left unclear. To the best of our knowledge no prior work has provided an exact semantics for general-purpose declarative continuous queries over streams and relations.

It may appear initially that the problem is not a difficult one: We take a relational query language, replace references to relations with references to streams, register the query with the stream processor, and wait for answers to arrive. For simple monotonic queries over complete stream histories indeed this approach is nearly sufficient. However, as queries get more complex—when we add aggregation, subqueries, windowing constructs, relations mixed with streams, etc.—the situation becomes much murkier. Even a simple query such as:

```
Select * From S[Rows 5], R
Where S.A = R.B
```

[*] This work was supported by the National Science Foundation under grants IIS-0118173 and IIS-9817799, and by a 3Com Stanford Graduate Fellowship.

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 1–19, 2004.

where S is a stream, R is a relation, and `[Rows 5]` specifies a *sliding window*, has no single obvious interpretation that we know of.

In this paper we initially define an *abstract semantics* for continuous queries based on "black box" components—any relational query language, any window specification language, and a set of relation-to-stream operators. We then define a *concrete language* that instantiates the black boxes in our abstract semantics, and that we are implementing in the *STREAM* prototype (*STanford stREeam datA Manager*), a general-purpose Data Stream Management System (DSMS) being developed at Stanford [9]. Our concrete language also has been used to specify a wide variety of continuous queries in a public repository of data stream applications we are curating [10].

In defining our abstract semantics and concrete language we had certain goals in mind:

1. We wanted to exploit well-understood relational semantics to the extent possible.
2. We wanted queries performing simple tasks to be easy and compact to write. Conversely, we wanted simple-appearing queries to do what one expects.

We believe these goals have been achieved to a large extent. To summarize the contributions of this paper:

- We formalize streams, updateable relations, and their interrelationship (Section 4).
- We define an abstract semantics for continuous queries constructed from three building blocks: any relational query language to operate on relations, any window specification language to convert streams to relations, and a set of three operators that convert relations to streams (Section 5).
- We propose a concrete language, *CQL* (for *Continuous Query Language*), which instantiates the abstract semantics using SQL as its relational query language and a window specification language derived from SQL-99. We define syntactic shortcuts and defaults in CQL for convenient and intuitive query formulation. We also compare the expressiveness of CQL against related query languages (Section 6).

2 Related Work

A comprehensive description of work related to data streams and continuous queries is given in [11]. Here we focus on work related to languages and semantics for continuous queries.

Continuous queries have been used either explicitly or implicitly for quite some time. *Materialized views* [12] are a form of continuous query, since a view is continuously updated to reflect changes to its base relations. Reference [13] extends materialized views to include *chronicles*, which essentially are continuous data streams. Operators are defined over chronicles and relations to produce other chronicles, and also to transform chronicles to materialized views. The operators are constrained to ensure that the resulting materialized views can be maintained incrementally without referencing entire chronicle histories.

Continuous queries were introduced explicitly for the first time in *Tapestry* [14] with a SQL-based language called *TQL*. (A similar language is considered in [15].) Conceptually, a TQL query is executed once every time instant as a one-time SQL query over the snapshot of the database at that instant, and the results of all the one-time queries are merged using set union. Several systems use continuous queries for information dissemination, e.g., [4, 6, 16]. The semantics of continuous queries in these systems is also based on periodic execution of one-time queries as in Tapestry. In Section 6.4, we show how Tapestry queries and materialized views over relations and chronicles can be expressed in CQL.

The abstract semantics and concrete language proposed in this paper are more general than any of the languages above, incorporating window specifications, constructs for freely mixing and mapping streams and relations, and the full power of any relational query language. Recent work in the *TelegraphCQ* system [17] proposes a declarative language for continuous queries with a particular focus on expressive windowing constructs. The *TelegraphCQ* language is discussed again briefly in Section 7. The *ATLaS* [18] SQL extension provides language constructs for expressing incremental computation of aggregates over windows on streams, but in the context of simple SPJ queries. GSQL is a SQL-like language developed for *Gigascope*, a DSMS designed for network monitoring applications. GSQL queries can also be expressed in CQL. GSQL is discussed in greater detail in Section 6.4.

Several systems support procedural continuous queries, as opposed to the declarative approach in this paper. The *event-condition-action* rules of active database systems, closely related to SQL *triggers*, fall into this category [19]. The *Aurora* system [20] is based on users directly creating a network of stream operators. A large number of operator types are available, from simple stream filters to complex windowing and aggregation operators. The *Tribeca* stream-processing system for network traffic analysis [21] supports windows, a set of operators adapted from relational algebra, and a simple language for composing query plans from them. Tribeca does not support joins across streams. Both Aurora and Tribeca are compared against CQL in more detail in Section 6.4.

Since stream tuples have timestamps and therefore ordering, our semantics and query language are related to *temporal* [22] and *sequence* [23] query languages. In most respects the temporal or ordering constructs in those languages subsume the corresponding features in ours, making our language less expressive but easier to implement efficiently. Also note that the semantics of temporal and sequence languages is for one-time, not continuous, queries.

3 Running Example

We introduce a running example based on a fabricated online auction application originally proposed as part of a benchmark for data stream systems [24]. This particular running example has been selected (and purposely kept simple) not for its realism but in order to easily illustrate various aspects of our semantics and query language. The interested reader is referred to our public query repository [10] for more complex and realistic stream applications, including a large number and variety of queries expressed in the language proposed in this paper.

The online auction application consists of users, the transactions of the users, and continuous queries that users or administrators of the system register for various monitoring purposes. Before submitting any transaction to the system, a user registers by providing a name and current state of residence. Registered users can later deregister. Three kinds of transactions are available in the system: users can place an item for auction and specify a starting price for the auction, they can close an auction they previously started, and they can bid for currently active auctions placed by other users by specifying a bid price. From the point of view of the auction system, all user registrations, deregistrations, and transactions are provided in the form of a continuous unbounded data stream.

Users also can register various monitoring queries in the system. For example, a user might request to be notified about any auction placed by a user from California within a specified price range. The auction system itself can run continuous queries for administrative purposes, such as: (1) Whenever an auction is closed, generate an entry with the closing price of the auction based on bid history. (2) Maintain the current set of active auctions and currently highest bid for them (to support "ad-hoc" queries from potential future bidders). (3) Maintain the current top 100 "hot items," *i.e.*, 100 items with the most number of bids in the last hour.

We will formalize details of the running example as the paper progresses.

4 Streams and Relations

In this section we define a formal model of streams, relations, and mappings between them, which we will use as the basis for our abstract semantics in Section 5. (In Section 7 we justify our decision to have both streams and relations instead of a purely stream-based approach.) For now let us assume any global, discrete, ordered time domain $\mathcal{T}$, such as the nonnegative numbers. A *time instant* (or simply *instant*) is any value from $\mathcal{T}$. As in the standard relational model, each stream and relation has a fixed schema consisting of a set of attributes.

Definition 1 (**Stream**) A stream S is a bag of *elements* $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of the stream and $\tau \in \mathcal{T}$ is the *timestamp* of the element. □

Definition 2 (**Relation**) A relation R is a mapping from $\mathcal{T}$ to a finite but unbounded bag of tuples, where each tuple belongs to the schema of the relation. □

A stream element $\langle s, \tau \rangle \in S$ indicates that tuple s arrives on stream S at time τ. Note that the timestamp is not part of the schema of the stream. For a given time instant $\tau \in \mathcal{T}$ there could be zero, one, or multiple elements with timestamp τ in a stream S. However we require that there be a finite (but unbounded) number of elements with a given timestamp. In addition to the source data streams that arrive at a DSMS, streams may result from queries or subqueries as described in Section 5. We use the terms *base stream* and *derived stream* to distinguish between source streams and streams resulting from queries or subqueries. For a stream S, we use the phrase *elements of S up to τ* to denote the elements of S with timestamp $\leq \tau$.

A relation R defines an unordered bag of tuples at any time instant τ, denoted $R(\tau)$. Note the difference between this definition for relation and the standard one: In the usual relational model a relation is simply a set (or bag) of tuples, with no notion of time as far as the semantics of relational query languages are concerned. We use the term *instantaneous relation* to denote a relation in the traditional static bag-of-tuples sense, and *relation* to denote a time-varying bag of tuples as in Definition 2. In addition to the stored relations in a DSMS, relations may result from queries or subqueries as described in Section 5. We use the terms *base relation* and *derived relation* to distinguish between stored relations and relations resulting from queries or subqueries.

Note that the time domain $\mathcal{T}$ may be an application-defined notion of logical time—it need not relate to any notion of physical time. $\mathcal{T}$ need not even be of type *Datetime*—our only requirement is that $\mathcal{T}$ is discrete and ordered, and that the streams and relations involved in a single query all use the same domain $\mathcal{T}$. (Details of our approach to flexible time management in data stream systems appear in [25].)

4.1 Modeling the Running Example

We now formally model the running example introduced in Section 3, drawn from [24]. The input to the online auction system consists of the following five streams:

- `Register(user_id,name,state)`: An element on this stream denotes the registration of a user identified by `user_id`. A user may register more than once in order to change his state, but we assume in our example queries that users do not change their name (i.e., there is a functional dependency `user_id` → `name`).
- `Deregister(user_id)`: An element on this stream denotes the deregistration of a user identified by `user_id`. We assume that a user deregisters at most once, i.e., `user_id` is a key for the `Deregister` stream.
- `Open(item_id,seller_id,start_price)`: An element on this stream denotes the start of an auction on `item_id` by user `seller_id` at a starting price of `start_price`. We assume that `item_id` is a key for this stream.
- `Close(item_id)`: An element on this stream denotes the closing of the auction on `item_id`, and `item_id` is again a key.
- `Bid(item_id,bidder_id,bid_price)`: An element on this stream denotes user `bidder_id` registering a bid of price `bid_price` for the open auction on `item_id`. We assume bid prices for each item strictly increase over time and are all greater than the starting price.

The time domain for the running example is *Datetime*, and the timestamps of stream elements correspond to the logical time of occurrence of the registration, deregistration, or a transaction.

An example of a (derived) relation is the relation `User(user_id,name,state)` containing the currently registered users. This relation is derived from streams `Register` and `Deregister`, as will be shown in Query 6.4 of Section 6.3.

4.2 Mapping Operators

We consider three classes of operators over streams and relations: *stream-to-relation* operators, *relation-to-relation* operators, and *relation-to-stream* operators. We decided not to introduce *stream-to-stream* operators, instead requiring those operators to be composed from the other three types. One rationale for this decision is goal #1 from Section 1—exploiting relational semantics whenever possible. Other aspects of this decision are discussed in Section 7, after our semantics and language are formalized.

1. A *stream-to-relation operator* takes a stream S as input and produces a relation R with the same schema as S as output. At any instant τ, $R(\tau)$ should be computable from the elements of S up to τ.
2. A *relation-to-relation operator* takes one or more relations $R_1, \ldots, R_n$ as input and produces a relation R as output. At any instant τ, $R(\tau)$ should be computable from the states of the input instantaneous relations at τ, i.e., $R_1(\tau), \ldots, R_n(\tau)$.
3. A *relation-to-stream operator* takes a relation R as input and produces a stream S with the same schema as R as output. At any instant τ the elements of S with timestamp τ should be computable from $R(\tau')$ for $\tau' \leq \tau$. In fact for the three operators we introduce in Section 5.1 only $R(\tau)$ and $R(\tau - 1)$ are needed, where $\tau - 1$ generically denotes the time instant preceding τ in our discrete time domain $\mathcal{T}$.

Example 1. Consider the Bid stream from Section 4.1. Suppose we take a "sliding window" over this stream that contains the bids over the last ten minutes. This windowing operator is an example of a stream-to-relation operator: the output relation R at time τ contains all the bids in stream Bid with timestamp between τ and τ minus 10 minutes (in the *Datetime* domain).

Now consider the operator Avg(bid_price) over the relation R output from the window operator. This aggregation is an example of a relation-to-relation operator: at time τ it takes instantaneous relation $R(\tau)$ as input and outputs an instantaneous relation with one single-attribute tuple containing the average price of the bids in $R(\tau)$—that is, the average price of bids in the last ten minutes on stream Bid. Finally, a simple relation-to-stream operator might stream the average price resulting from operator Avg(bid_price) every time the average price changes. □

5 Abstract Semantics

In this section we present an abstract semantics for continuous queries using our model of streams, relations, and mappings among them from Section 4. We assume that a query is constructed from the following three building blocks:

1. Any relational query language, which we can view abstractly as a set of relation-to-relation operators as defined in Section 4.
2. A *window specification language*, which we can view as a set of stream-to-relation operators as defined in Section 4. In theory these operators need not have anything to do with "windows," but in practice windowing is the most common way of producing bounded sets of tuples from unbounded streams [11][1].

[1] Certain types of sampling [26] provide another way of doing so, and do fit into our framework. However, we are implementing sampling as a separate operator in our query language as outlined in [27].

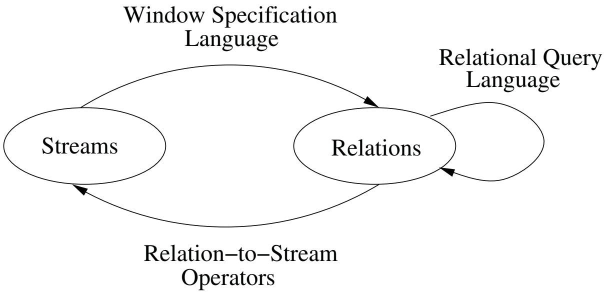

Window Specification Language
Relational Query Language
Streams
Relations
Relation–to–Stream Operators

Fig. 1. Mappings used in abstract semantics.

3. Three relation-to-stream operators: `Istream`, `Dstream`, and `Rstream`, defined shortly in Section 5.1. Here we could certainly be more abstract—any relation-to-stream language could be used—but the three operators we define capture the required expressiveness for all queries we have considered [10].

The interaction among these three building blocks is depicted in Figure 1. With these building blocks, our abstract semantics is straightforward: A well-formed continuous query is simply assembled in a type-consistent way from streams, relations, and the operators in Figure 1. Specifically, consider a time instant τ. If a derived relation R is the output of a window operator over a stream S, then $R(\tau)$ is computed by applying the window semantics on the elements of S up to τ. If a derived relation R is the output of a relational query, then $R(\tau)$ is computed by applying the semantics of the relational query on the input relations at time τ. Finally, any derived stream is the output of an `Istream`, `Dstream`, or `Rstream` operator over a relation R and is computed as described next in Section 5.1. The final result of the continuous query can be a relation or a stream.

In the remainder of this section we define our three relation-to-stream operators, then provide a "concrete" example of our abstract semantics. Further examples are provided later in the paper in the context of our concrete language.

5.1 Relation-to-Stream Operators

As mentioned earlier, our abstract semantics could assume a "black box" language for mapping relations to streams, as we have done for streams-to-relations. However, at this point we will formalize three operators that we have found to be particularly useful and, so far, sufficient [10]. Note that operators $\cup$, $\times$, and $-$ below are assumed to be the bag versions.

- `Istream` (for "insert stream") applied to relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau) - R(\tau - 1)$. Using 0 to denote the earliest instant in time domain $\mathcal{T}$, and assuming $R(-1) = \phi$ for notational simplicity, we have:

$$\texttt{Istream}(R) = \underset{\tau \geq 0}{\quad} ((R(\tau) - R(\tau - 1)) \times \{\tau\})$$

- Analogously, `Dstream` (for "delete stream") applied to relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau - 1) - R(\tau)$. Formally:

$$\texttt{Dstream}(R) = \underset{\tau>0}{\quad} ((R(\tau-1) - R(\tau)) \times \{\tau\})$$

- Rstream (for “relation stream”) applied to relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in R at time τ. Formally:

$$\texttt{Rstream}(R) = \underset{\tau\geq 0}{\quad} (R(\tau) \times \{\tau\})$$

With some basic windowing and relational constructs such as those defined for CQL in Section 6, Rstream subsumes the combination of Istream and Dstream. For many queries Istream is more natural to use than Rstream, and Dstream is a natural counterpart to Istream. Hence we decided to introduce all three operators in keeping with goal #2 from Section 1—simple queries should be easy and compact to write. (The astute reader may note that Dstream does not appear again in this paper, however it does occasionally have its uses, as seen in [10].)

5.2 Example

Using our abstract semantics let us revisit the example continuous query in Section 1 (expressed there as SQL), to understand its behavior in detail and to illustrate the semantics of our relation-to-stream operators. Using relational algebra as our relational query language and $S[n]$ to denote an n-tuple sliding window over stream S, the query can be rewritten as:

$$S[5] \bowtie_{S.A=R.B} R$$

At any time instant τ, $S[5]$ produces an instantaneous relation containing the last five tuples in S up to τ. That relation is joined with $R(\tau)$, producing the final query result which in this case is a relation. The result relation may change whenever a new tuple arrives in S or R is updated.

Now suppose we add an outermost Istream operator to this query, converting the relational result into a stream. With Istream semantics, a new element $\langle u, \tau \rangle$ is streamed whenever tuple u is inserted into $S[5] \bowtie R$ at time τ as the result of a stream arrival or relation update. Note however that u must not be present in the join result at time $\tau - 1$, which creates an unusual but noteworthy subtlety in Istream semantics: If S has no key, and it contains two identical tuples s_i and s_{i+5} that are five tuples apart and that join with a tuple $r \in R$, then the Istream operator will not produce tuple (s_{i+5}, r) in the result stream when s_{i+5} arrives on S since the relational output of the join is unchanged. Replacing the outermost Istream operator with Rstream will produce element $\langle (s_{i+5}, r), \tau \rangle$ in the result stream, however now the entire join result will be streamed at each instant of time.

Fortunately, anomalous queries such as the example above are unusual and primarily of theoretical interest. For example, no such queries have arisen in [10], and we were unable to construct a realistic concrete example for this paper.

6 Concrete Query Language

Our concrete language *CQL* (standing for *Continuous Query Language* and pronounced “C-Q-L” or “sequel,” depending on taste) uses SQL as its relational query language. In theory it supports all SQL constructs, but our current implementation eliminates many of the more esoteric ones. CQL contains three syntactic extensions to SQL:

1. Anywhere a relation may be referenced in SQL, a stream may be referenced in CQL.
2. In CQL every reference to a base stream, and every subquery producing a stream, must be followed immediately by a window specification. Our window specification language is derived from SQL-99 and defined in Section 6.1. Syntactic shortcuts based on default windows are defined in Section 6.2.
3. In CQL any reference to a relation, or any subquery producing a relation, may be converted into a stream by applying any of the operators `Istream`, `Dstream`, or `Rstream` defined in Section 5.1 to the entire `Select` list. Default conversions are applied in certain cases; see Section 6.2.

After defining our window specification language in Section 6.1, we introduce CQL’s syntactic shortcuts and defaults in Section 6.2. We provide detailed examples of CQL queries in Section 6.3, and in Section 6.4 we compare CQL against related query languages.

Although we do not specify them explicitly as part of our language, incorporating user-defined functions, user-defined aggregates, or user-defined window operators poses no problem in CQL, at least from the semantic perspective.

6.1 Window Specification Language

Currently CQL supports primarily *sliding* windows, and it supports three types: time-based, tuple-based, and partitioned. Other types of sliding windows,*fixed* windows [21], *tumbling* windows [20], *value-based* windows [23], or any other windowing construct can be incorporated into CQL easily—new syntax must be added, but the semantics of incorporating a new window type relies solely on the semantics of the window operator itself, thanks to our “building-blocks” approach.

In the definitions below note that a stream S may be a base stream or a derived stream produced by a subquery.

Time-Based Windows. A time-based sliding window on a stream S takes a time-interval T as a parameter and is specified by following S in the query with `[Range T]`[2]. Intuitively, this window defines its output relation over time by sliding an interval

[2] In all three of our window types we dropped the keyword `Preceding` appearing in the SQL-99 syntax and in our earlier specification [27]—we only have “preceding” windows for now so the keyword is superfluous. Also we do not specify a syntax, type, or restrictions for time-interval T at this point. Currently our system is restricted to use the *Datetime* type for timestamps and the SQL-99 standard for time intervals. Examples are given in Section 6.3.

of size T time units over S. More formally, the output relation R of "`S [Range T]`" is defined as:

$$R(\tau) = \{s \mid \langle s, \tau' \rangle \in S \ \wedge \\ (\tau' \leq \tau) \ \wedge \ (\tau' \geq \max\{\tau - T, 0\})\}$$

Two important special cases are $T = 0$ and $T = \infty$. When $T = 0$, $R(\tau)$ consists of tuples obtained from elements of S with timestamp τ. In CQL we introduce the syntax "`S [Now]`" for this special case. When $T = \infty$, $R(\tau)$ consists of tuples obtained from all elements of S up to τ and uses the SQL-99 syntax "`S [Range Unbounded]`."

Tuple-Based Windows. A tuple-based sliding window on a stream S takes a positive integer N as a parameter and is specified by following S in the query with `[Rows N]`. Intuitively, this window defines its output relation by sliding a window of size N tuples over S. More formally, for the output relation R of "`S [Rows N]`," $R(\tau)$ consists of tuples obtained from the N elements with the largest timestamps in S no greater than τ (or all elements if the length of S up to τ is $\leq N$). Suppose we specify a sliding window of N tuples and at some point there are several tuples with the Nth most recent timestamp (while for clarity let us assume the other $N - 1$ more recent timestamps are unique). Then we must "break the tie" in some fashion to generate exactly N tuples in the window. We assume such ties are broken arbitrarily. Thus, tuple-based sliding windows may be nondeterministic—and therefore may not be appropriate—when timestamps are not unique. The special case of $N = \infty$ is specified by `[Rows Unbounded]`, and is equivalent to `[Range Unbounded]`.

Partitioned Windows. A partitioned sliding window on a stream S takes a positive integer N and a subset $\{A_1, \ldots, A_k\}$ of S's attributes as parameters. It is specified by following S in the query with `[Partition By` A_1`,...,`A_k `Rows N]`. Intuitively, this window logically partitions S into different substreams based on equality of attributes $A_1, \ldots, A_k$ (similar to SQL `Group By`), computes a tuple-based sliding window of size N independently on each substream, then takes the union of these windows to produce the output relation. More formally, a tuple s with values $a_1, \ldots, a_k$ for attributes $A_1, \ldots, A_k$ occurs in output instantaneous relation $R(\tau)$ *iff* there exists an element $\langle s, \tau' \rangle \in S$ such that $\tau' \leq \tau$ is among the N largest timestamps among elements whose tuples have values $a_1, \ldots, a_k$ for attributes $A_1, \ldots, A_k$. Note that analogous time-based partitioned windows would provide no additional expressiveness over nonpartitioned time-based windows.

6.2 Syntactic Shortcuts and Defaults

In keeping with goal #2 in Section 1, we permit some syntactic "shortcuts" in CQL that result in the application of certain defaults. Of course there may be cases where the default behavior is not what the author intended, so we assume that when queries are registered the system informs the author of the defaults applied and offers the opportunity to edit the expanded query. There are two classes of shortcuts: omitting window specifications (Section 6.2) and omitting relation-to-stream operators (Section 6.2).

Default Windows. When a base stream or a stream derived from a subquery is referenced in a CQL query and is not followed by a window specification, an Unbounded window is applied by default. (Recall from the beginning of this section that every reference to a stream within a query must be followed immediately by a window specification.) While the default Unbounded window usually produces appropriate behavior, there are cases where a Now window is more appropriate, e.g., when a stream is joined with a relation; see Query 6.5 in Section 6.3 for an example. Also Unbounded windows often may be replaced by Now windows as a query rewrite optimization; see Query 6.1 in Section 6.3.

Default Relation-to-Stream Operators. There are two cases in which it seems natural for authors to omit an intended Istream operator from a CQL query:

1. On the outermost query, even when *streamed results* rather than *stored results* are desired [27].
2. On an inner subquery, even though a window is specified on the subquery result.

For the first case we add an Istream operator by default whenever the query produces a relation that is *monotonic*. A relation R is monotonic *iff* $R(\tau_1) \subseteq R(\tau_2)$ whenever $\tau_1 \leq \tau_2$. A conservative monotonicity test can be performed statically. For example, a base relation is monotonic if it is known to be append-only, "S [Range Unbounded]" is monotonic for any stream S, and the join of two monotonic relations also is monotonic. If the result of a CQL query is a monotonic relation then it makes intuitive sense to convert the relation into a stream using Istream. If it is not monotonic, the author might intend Istream, Dstream, or Rstream, so we do not add a relation-to-stream operator by default.

For the second case we add an Istream operator by default whenever the subquery is monotonic. If it is not, then the intended meaning of a window specification on the subquery result is somewhat ambiguous, so a semantic (type) error is generated, and the author must add an explicit relation-to-stream operator.

6.3 Example Queries

We present several example queries to illustrate the syntax and semantics of CQL. All queries are based on the online auction schema introduced in Section 3 and formalized in Section 4.1.

Query 6.1: Stream Filter. *Select auctions where the starting price exceeds 100 and produce the result as a stream.*

```
Select * From Open Where start_price > 100
```

This query relies on two CQL defaults. Since the Open stream is referenced without a window specification, an Unbounded window is applied by default. At time τ, the relational result of the unbounded window contains tuples from all elements of Open up to τ, and the output relation of the entire query contains the subset of those tuples that satisfy the filter predicate. Since the output relation is monotonic, a default Istream operator is applied, converting the output relation into a stream consisting of each element of Open that satisfies the filter predicate.

This query can also be written using a Now window instead of the (default) Unbounded window and Rstream instead of the (default) Istream:

```
Select Rstream(*) From Open[Now]
Where start_price > 100
```

The rewritten query suggests a more efficient execution strategy.

Query 6.2: Sliding-Window Aggregate. *Maintain a running count of the number of bids in the last hour on items with* item_id *in the range 100 to 200.*

```
Select Count(*) From Bid[Range 1 Hour]
Where item_id >= 100 and item_id <= 200
```

The Bid stream has an explicit window specification and the result of the query is a nonmonotonic singleton relation, so no defaults are applied. If the author adds an Istream operator, then the result will instead stream a new value each time the count changes. If the count should be streamed at each time instant regardless of whether its value has changed, then an Rstream operator should be used instead of Istream.

Query 6.3: Stream Subquery. *Maintain a table of the currently open auctions.*

```
Select * From Open
Where item_id Not In (Select * From Close)
```

Unbounded windows are applied by default on both Open and Close. The subquery in the Where clause returns a monotonic relation containing all closed auctions at any time, but a default Istream operator is not applied since there is no window specification following the subquery. The relational result of the entire query is not monotonic—auction tuples are deleted from the result when the auction is closed—and therefore an outermost Istream operator is not applied.

Query 6.4: Derived Relation. *Compute the* User *relation (described in Section 4.1) containing the currently registered users.*

```
Select user_id, name, state
From Register[Partition By user_id Rows 1]
Where user_id Not In
              (Select * From Deregister)
```

The partitioned window on the Register stream obtains the latest registration for each user, and the Where clause filters out users who have already deregistered. In subsequent examples we refer to User in our queries as a normal relation. The query above defining relation User can be substituted syntactically into examples referencing User, although in a system we might instead choose to maintain User as a materialized view.

Query 6.5: Relation-Stream Join. *Output as a stream the auctions started by users who were California residents when they started the auction.*

```
Select Istream(Open.*) From Open[Now], User
Where seller_id = id and state = 'CA'
```

This query joins the Open stream with the User relation. The Now window on Open ensures that a stream tuple joins with the corresponding User tuple to find the state of residence at the time the auction starts. If we used an Unbounded window on Open instead of the Now window, then whenever a user moved into California all previous auctions started by that user would be generated in the result stream.

This query is an example where if a window specification were omitted the default Unbounded window would not provide the intended behavior. In general, if a stream is joined with a relation in order to add attributes to or filter the stream, then a Now window on the stream coupled with an Istream or Rstream operator usually provides the desired behavior. However, this rule is not so hard-and-fast that it would be appropriate to identify these cases and apply a default Now window.

Query 6.6: Windowed-Stream Join. *Stream the* item_id*'s for all auctions closed within 5 hours of their opening.*

```
Select Istream(Close.item_id)
From Close[Now], Open[Range 5 Hours]
Where Close.item_id = Open.item_id
```

This query streams any item_id from Close whose corresponding Open tuple arrived within the last 5 hours. Note that by our abstract semantics defined in Section 5, the timestamps on result stream elements correspond to the timestamps on the Close stream elements from which they are produced.

Query 6.7: Complex from Clause. *Compute the closing price of each auction and stream the result.*

We include the possibility of auctions with no bids, and solely for purposes of illustration we stream the starting price of an auction with no bids as its closing price. Recall that we assume bid prices are strictly increasing over time.

```
Select Istream(Close.item_id, P.price)
From Close[Now],
     ((Select item_id, bid_price as price
       From Bid)
       Union
      (Select item_id, start_price as price
       From Open))
     [Partition By item_id Rows 1] As P
Where Close.item_id = P.item_id
```

Unbounded windows are applied by default on the Bid and Open streams, however after we take their union the partitioned window extracts the latest element for each item_id. An Istream operator is (and needs to be) applied to the Union result by default, since the relational output of the Union subquery is monotonic and is followed by a window specification[3].

Recall that we assume bids are not permitted once an auction closes, so new tuples are produced in the join only when they are produced on stream Close. The new join result tuple contains the latest transaction for the closed auction—either the latest bid or the opening of the auction—from which the closing price is extracted. Finally observe that based on application semantics the author could have used an Unbounded window on stream Close instead of a Now window and the query result would be equivalent, so the default window is acceptable for all three streams in this query.

Monotonicity. Queries 6.1 and 6.7 exploited monotonicity, which was relatively straightforward to detect in both cases. We anticipate that fairly simple (conservative) monotonicity tests can be used in general, since failing to detect monotonicity when it holds does not pose a real problem: For a monotonic subquery with a window operator, the author would be notified of the type inconsistency and would need to add the Istream operator explicitly. Similarly, if a default Istream is not applied to the outermost query the author would know that the query returns a relation and can add the Istream operator as desired.

6.4 Comparison with Other Languages

Now that we have presented our language we can provide a more detailed comparison against some of the related languages for continuous queries over streams and relations that were discussed briefly in Section 2. Specifically, we show that basic CQL (without user-defined functions, aggregates, or window operators) is strictly more expressive than *Tapestry* [14], *Tribeca* [21], and materialized views over relations with or without *chronicles* [13]. We also discuss *Aurora* [20], although it is difficult to compare CQL against Aurora because of Aurora's graphical, procedural nature. The query language of TelegraphCQ [17] is discussed in Section 7.

Views and Chronicles. Any conventional materialized view defined using a SQL query Q can be expressed in CQL using the same query Q with CQL semantics.

The *Chronicle Data Model* (*CDM*) [13] defines chronicles, relations, and persistent views, which are equivalent to streams, base relations, and derived relations in our terminology. For consistency we use our terminology instead of theirs. CDM supports two classes of operators based on relational algebra, both of which can be expressed in CQL. The first class takes streams and (optionally) base relations as input and and produces streams as output. Each operator in this class can be expressed equivalently in CQL by applying a Now window on the input streams, translating the relational algebra

[3] Technically we have not specified syntax in this paper for applying Istream to the result of a Union of two subqueries. In practice we allow Istream, Dstream, or Rstream to be placed outside of any query producing a relation, which for SPJ queries is equivalent to applying it to the Select list.

operator to SQL, and applying an `Rstream` operator to produce a streamed result. For example, join query $S \bowtie_{S.A=S'.B} S'$ in CDM is equivalent to the CQL query:

```
Select Rstream(*) From S[Now], S'[Now]
Where S.A = S'.B
```

The second class of operators take a stream as input and produce a derived relation as output. These operators can be expressed in CQL by applying an `Unbounded` window on the input stream and translating the relational algebra operator to SQL.

The operators in CDM are strictly less expressive than CQL. CDM does not support sliding windows over streams, although it has implicit `Now` and `Unbounded` windows as described above. Furthermore, CDM distinguishes between base relations, which can be joined with streams, and derived relations (persistent views), which cannot. These restrictions ensure that derived relations in CDM can be maintained incrementally in time logarithmic in the size of the derived relation. CQL queries, on the other hand, could require unbounded time and memory, as we have shown in [7] and addressed in [27].

Tapestry. Tapestry queries [14] are expressed using SQL syntax. At time τ, the result of a Tapestry query Q contains the set of tuples logically obtained by executing Q as a relational SQL query at every instant $\tau' \leq \tau$ and taking the set-union of the results. This semantics for Q is equivalent to the CQL query:

```
Select Istream(Distinct *)
From (Istream(Q))[Range Unbounded]
```

Tapestry does not support sliding windows over streams or any relation-to-stream operators.

Tribeca. Tribeca is based on a set of stream-to-stream operators and we have shown that all of the Tribeca operators specified in [21] can be expressed in CQL; details are omitted. Two of the more interesting operators are `demux` (demultiplex) and `mux` (multiplex). In a Tribeca query the `demux` operator is used to split a single stream into an arbitrary number of substreams, the substreams are processed separately using other (stream-to-stream) operators, then the resulting substreams are merged into a single result stream using the `mux` operator. This type of query is expressed in CQL using a combination of partitioned window and `Group By`.

Like chronicles and Tapestry, Tribeca is strictly less expressive than CQL. Tribeca queries take a single stream as input and produce a single stream as output, with no notion of relation. CQL queries can have multiple input streams and can freely mix streams and relations.

Aurora. Aurora queries are built from a set of eleven operator types. Operators are composed by users into a global query execution plan via a “boxes-and-arrows” graphical interface. It is somewhat difficult to compare the procedural query interface of Aurora against a declarative language like CQL, but we can draw some distinctions.

The aggregation operators of Aurora (*Tumble*, *Slide*, and *XSection*) are each defined from three user-defined functions, yielding nearly unlimited expressive power. The aggregation operators also have optional parameters related to system and application time (see Section 4 and [25]). For example, these parameters can direct the operator to take certain action if no stream element has arrived for T seconds, making the semantics dependent on stream arrival rates and nondeterministic, an approach we have not considered to date in CQL.

All operators in Aurora are stream-to-stream, and Aurora does not explicitly support relations. Therefore, in order to express CQL queries involving derived relations and relation-to-relation operators, Aurora procedurally manipulates state corresponding to a derived relation.

Gigascope Query Language (GSQL). *GSQL* is a SQL-like query language developed for *Gigascope*, a DSMS designed specifically for network monitoring applications [28]. Like Aurora, GSQL is a stream-only language, so relations must be created and manipulated using user-defined functions. Over streams GSQL's primary operators are selection, join, aggregation, and *merge*. Constraints on join and aggregation ensure that they are nonblocking: a join operator must contain a predicate involving an "ordered" attribute from each of the joining streams, and an aggregation operator must have at least one grouping attribute that is ordered. (Ordered attributes are generalizations of CQL timestamps.)

The four primary operations in GSQL can be expressed in CQL: Selection is straightforward. The GSQL merge operator can be expressed using `Union` in CQL. The GSQL join operator translates to a sliding-window join with an `Istream` operator in CQL. Finally, although it is nontrivial to express GSQL aggregation in CQL (requiring grouping and aggregation, projection, and join), it always is expressible; details are omitted.

7 Stream-Only Query Language

Our abstract semantics and therefore CQL distinguish two fundamental data types, relations and streams. We can derive a stream-only language, L_s, from our language L as follows.

- Corresponding to each n-ary relation-to-relation operator O in L, there is an n-ary stream-to-stream operator O_s in L_s. The semantics of $O_s(S_1, \ldots, S_n)$ when expressed in L is $\texttt{Rstream}(O(S_1[\texttt{Now}], \ldots, S_n[\texttt{Now}]))$.
- Corresponding to each window operator W in L, there is a unary stream-to-stream operator W_s in L_s. The semantics of $S[W_s]$ when expressed in L is $\texttt{Rstream}(S[W])$.
- There are no operators in L_s corresponding to relation-to-stream operators of L.

It can be shown that L and L_S have essentially the same expressive power. Clearly any query in L_S can be rewritten in L. Given a query Q in L, we obtain a query Q_S in L_S by performing the following three steps. First, transform Q to an equivalent query Q' that has `Rstream` as its only relation-to-stream operator (this step is always possible as indicated in Section 5.1). Second, replace every input relation R_i in Q' with

Rstream(R_i). Finally, replace every relation-to-relation and window operator in Q with its L_S equivalent according to the definitions above. As it turns out, the language L_s is quite similar to the stream-to-stream approach being taken in *TelegraphCQ* [17].

We chose our dual approach over the stream-only approach for at least three reasons. First is goal #1 from Section 1: exploiting well-known relational semantics to the extent possible. Second, our experience with a large number of queries [10] suggests that the dual approach results in more intuitive queries than the stream-only approach. Third, having both relations and streams cleanly generalizes materialized views, as discussed in detail in Section 6.4. Note that the *Chronicle Data Model* [13] also takes an approach similar to ours—it supports both chronicles (closely related to streams) and materialized views (relations). The Chronicle Data Model was also discussed in detail in Section 6.4.

8 Conclusion

To the best of our knowledge this paper is one of the first to provide an exact semantics for general-purpose declarative continuous queries over streams and relations. We first presented an abstract semantics based on any relational query language, any window specification language to map from streams to relations, and a set of operators to map from relations to streams. We then proposed CQL, a concrete language that instantiates the "black boxes" in our abstract semantics using SQL as the relational query language and window specifications derived from SQL-99. We showed how CQL encompasses several previous languages and semantics for continuous queries in terms of expressiveness.

We are implementing CQL as part of a general-purpose Data Stream Management System at Stanford. Some details of our overall system and our approach to query processing are provided in [27]. At the time of writing (Summer 2003) a significant portion of CQL is running [9].

Acknowledgments

We are grateful to Brian Babcock for his first heroic attempts at codifying a semantics and thereby stimulating this work in his absence, to Keith Ito and Itaru Nishizawa for their efforts implementing CQL, and to the entire STREAM group at Stanford for many stimulating discussions.

References

1. Golab, L., Ozsu, T.: Issues in data stream management. SIGMOD Record **32** (2003) 5–14
2. Gehrke (ed.), J.: Special issue on data stream processing. IEEE Data Engineering Bulletin **26** (2003)
3. Babu, S., Widom, J.: Continuous queries over data streams. SIGMOD Record **30** (2001) 109–120
4. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A scalable continuous query system for internet databases. In: Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data. (2000) 379–390

5. Madden, S., Shah, M., Hellerstein, J., Raman, V.: Continuously adaptive continuous queries over streams. In: Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data. (2002) 49–60
6. Liu, L., Pu, C., Tang, W.: Continual queries for internet scale event-driven information delivery. IEEE Trans. on Knowledge and Data Engineering **11** (1999) 583–590
7. Arasu, A., Babcock, B., Babu, S., McAlister, J., Widom, J.: Characterizing memory requirements for queries over continuous data streams. In: Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems. (2002) 221–232
8. Chandrasekaran, S., Franklin, M.: Streaming queries over streaming data. In: Proc. 28th Intl. Conf. on Very Large Data Bases. (2002)
9. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., Widom, J.: STREAM: The Stanford Stream Data Manager. In: Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data. (2003) 665 Demo description.
10. : (SQR – A Stream Query Repository) `http://www.db.stanford.edu/stream/sqr`.
11. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proc. of the 2002 ACM Symp. on Principles of Database Systems. (2002) 1–16
12. Gupta, A., Mumick, I.S.: Maintenance of materialized views: Problems, techniques, and applications. IEEE Data Engineering Bulletin **18** (1995) 3–18
13. Jagadish, H., Mumick, I., Silberschatz, A.: View maintenance issues for the Chronicle data model. In: Proc. of the 1995 ACM Symp. on Principles of Database Systems. (1995) 113–124
14. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. In: Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data. (1992) 321–330
15. Barbara, D.: The characterization of continuous queries. Intl. Journal of Cooperative Information Systems **8** (1999) 295–323
16. Nguyen, B., Abiteboul, S., Cobena, G., Preda, M.: Monitoring XML data on the web. In: Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data. (2001) 437–448
17. Chandrasekaran, S., et al.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: Proc. of the 2003 Conf. on Innovative Data Systems Research. (2003) 269–280
18. Wang, H., Zaniolo, C., Luo, R.: ATLaS: A Turing-complete extension of SQL for data mining applications and streams (2002) Manuscript available at `http://wis.cs.ucla.edu/publications.html`.
19. Paton, N., Diaz, O.: Active database systems. ACM Computing Surveys **31** (1999)
20. Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.: Monitoring streams–a new class of data management applications. In: Proc. 28th Intl. Conf. on Very Large Data Bases. (2002) Material augmented by personal communication.
21. Sullivan, M.: Tribeca: A stream database manager for network traffic analysis. In: Proc. of the 1996 Intl. Conf. on Very Large Data Bases. (1996) 594
22. Ozsoyoglu, G., Snodgrass, R.: Temporal and real-time databases: A survey. IEEE Trans. on Knowledge and Data Engineering **7** (1995) 513–532
23. Seshadri, P., Livny, M., Ramakrishnan, R.: SEQ: a model for sequence databases. In: Proc. of the 1995 Intl. Conf. on Data Engineering. (1995) 232–239
24. Tucker, P.A., Tufte, K., Papadimos, V., Maier, D.: NEXMark – a benchmark for querying data streams (2002) Manuscript available at `http://www.cse.ogi.edu/dot/niagara/NEXMark/`.
25. Srivastava, U., Widom, J.: Flexible time management in data stream systems. Technical report, Stanford University Database Group (2003) Available at `http://dbpubs.stanford.edu/pub/2003-40`.

26. Vitter, J.: Random sampling with a reservoir. ACM Trans. on Mathematical Software **11** (1985) 37–57
27. Motwani, R., Widom, J., et al.: Query processing, resource management, and approximation in a data stream management system. In: Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR). (2003) 245–256
28. Cranor, C.D., Johnson, T., Spatscheck, O., Shkapenyuk, V.: Gigascope: A Stream Database for Network Applications. In: Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data. (2003) 647–651

XPath Query Processing*

Georg Gottlob[1] and Christoph Koch[1,2]

[1] DBAI, Technische Universität Wien, A-1040 Vienna, Austria
{gottlob,koch}@dbai.tuwien.ac.at
[2] LFCS, University of Edinburgh, Edinburgh EH9 3JZ, UK

Abstract. XPath 1 [4] is a practical language for selecting nodes from XML document trees and plays an essential role in other XML-related technologies such as XSLT and XQuery. Implementations of XPath need to scale well *both* with respect to the size of the XML data and the growing size and intricacy of the queries (i.e., *combined complexity*). Unfortunately, current XPath engines use query evaluation techniques that require time *exponential* in the size of queries in the worst case [1].
The main aim of this tutorial is to show *that* and *how* XPath can be processed much more efficiently. We present an algorithm for this problem with polynomial-time combined query evaluation complexity. Then we discuss various improvements over the basic evaluation algorithm, such as the context-restriction technique of [3], which lead to better worst-case bounds. We provide an overview over XPath fragments that can be processed yet more efficiently, most prominently one that can be evaluated in linear time (cf. [1, 3]). Next we discuss the parallel complexity of XPath. While full XPath is P-complete w.r.t. combined complexity, various (minor) restrictions are known which reduce the complexity to highly parallelizable complexity classes [2]. Finally, we give an overview of further recent work on efficient XPath processing, in particular using logic- and automata-based techniques.

References

1. G. Gottlob, C. Koch, and R. Pichler. "Efficient Algorithms for Processing XPath Queries". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, 2002.
2. G. Gottlob, C. Koch, and R. Pichler. "The Complexity of XPath Query Processing". In *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'03)*, San Diego, California, 2003.
3. G. Gottlob, C. Koch, and R. Pichler. "XPath Query Evaluation: Improving Time and Space Efficiency". In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE'03)*, Bangalore, India, Mar. 2003.
4. World Wide Web Consortium. XML Path Language (XPath) Recommendation. http://www.w3c.org/TR/xpath/, Nov. 1999.

[*] This tutorial is based on joint work with Reinhard Pichler and was partially supported by the Austrian Science Fund (FWF) under project No. Z29-N04 and the GAMES Network of Excellence of the European Union. The second author's work was sponsored by Erwin Schrödinger Grant J2169 of the FWF. Further information, including the slides of this tutorial, can be found at http://www.xmltaskforce.com.

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, p. 20, 2004.

Satisfiability of XPath Expressions

Jan Hidders

University of Antwerp
Dept. of Mathematics and Computer Science
Middelheimlaan 1, BE-2020 Antwerp, Belgium
jan.hidders@ua.ac.be

Abstract. In this paper, we investigate the complexity of deciding the satisfiability of XPath 2.0 expressions, i.e., whether there is an XML document for which their result is nonempty. Several fragments that allow certain types of expressions are classified as either in PTIME or NP-hard to see which type of expression make this a hard problem. Finally, we establish a link between XPath expressions and partial tree descriptions which are studied in computational linguistics.

1 Introduction

XPath is a simple language for selecting a set of nodes in an XML tree and as such it is used in many other XML-related standards such as XSLT, XQuery, XML Schema, XLink and XPointer. The satisfiability problem for XPath expressions is relevant for all these applications because it allows the detection of expressions that are probably erroneous and query optimizations that remove expressions that always return an empty result.

The satisfiability problem is a special case of the containment problem which has already been studied quite extensively [1, 12, 13, 16, 17]. However, in these studies usually only fragments with forward axes are considered, in which case most expressions are trivially satisfiable. Therefore, we will consider fragments in which all axes are allowed, including those that depend upon document order. We give four small examples of conflicts that may then occur:

1. `self::a/self::b`
   This path looks for a node that has at the same time name `a` and `b`.
2. `child::a/child::*/parent::b`
   This path requires that the node in the result has name `a` and `b`.
3. `/child::*/parent::*/parent::*`
   This path looks for a parent of the root node, which cannot exist.
4. `/preceding::*`
   This path looks for a node that precedes the root, but such a node cannot exist since the root is always the first node in the document order.

The organization of this paper is as follows. The next section contains the definition of XPath expressions and their semantics. Section 3 introduces the notion of tree description graph which are a special case of partial tree descriptions

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 21–36, 2004.

as studied in computational linguistics. Section 4 discusses the relationship between these tree description graphs and XPath expressions. Section 5 introduces a specific string matching problem that will be used in the following sections to show NP-hardness. Section 6 presents some complexity lowerbounds for certain fragments of XPath and Section 7 presents some upperbounds. Finally, Section 8 summarizes and discusses the presented results.

2 Initial Definitions

We start with the definition of the data model which is a simplification and abstraction of the full XML data model [8] and restricts itself to the element nodes. For this and following definitions we postulate an infinite set of tag names Σ.

Definition 1 (XML Tree). *An* XML tree *is a tuple* $T = (N, \lhd, r, \lambda, \prec)$ *such that* $(N, \lhd)$ *is a finite directed graph where* N *represents the set of* element nodes *and* $\lhd$ *represents the* parent-child relationship *that defines a tree with root* r*,* $\lambda : N \to \Sigma$ *is a labeling of the nodes that gives the tag name of each node and* $\prec$ *is a strict total order*[1] *over* N *that represents the* document order *and defines a pre-order tree-walk, i.e.,*

PTW1 *every child is smaller than its parent, i.e., if* $n_1 \lhd n_2$ *then* $n_1 \prec n_2$ *for all* $n_1, n_2 \in N$*, and*

PTW2 *if two nodes are siblings then all descendants of the smaller sibling are smaller than the larger sibling, i.e., for all two nodes* $n_1, n_2 \in N$ *for which there is a node* $n_3 \in N$ *such that* $n_3 \lhd n_1$ *and* $n_3 \lhd n_2$ *it holds that if* $n_1 \prec n_2$ *and* $n_1 \lhd^+ n_4$ *then* $n_4 \prec n_2$*, where* $\lhd^+$ *denotes the transitive closure of* $\lhd$*.*

In the following we let $\lhd^+$ denote the transitive closure of $\lhd$, and $\lhd^*$ the transitive and reflexive closure of $\lhd$. Next, we define the set of XPath expressions that we will consider. We will use a syntax in the style of [2] that abstracts from the official syntax [3] and is more suitable for formal presentations.

Definition 2 (XPath Expression). *The set of* XPath expressions *is defined by the following abstract grammar:*

$$\begin{aligned} P ::= {} & \epsilon \mid \uparrow \mid \downarrow \mid \uparrow^* \mid \downarrow^* \mid \dot{\twoheadleftarrow} \mid \dot{\twoheadrightarrow} \mid \\ & \Uparrow \mid \Sigma \mid P/P \mid P[P] \mid \\ & P \cap P \mid P \cup P \mid P - P \end{aligned}$$

where ϵ *represents the empty path or* self *axis,* $\uparrow$ *and* $\downarrow$ *represent the* parent *and* child *axis,* $\uparrow^*$ *and* $\downarrow^*$ *represent the* ancestor-or-self *and* descendant-or-self *axis,* $\dot{\twoheadleftarrow}$ *and* $\dot{\twoheadrightarrow}$ *represent the* preceding-sibling *and* following-sibling *axis,* $\Uparrow$ *represents the document root,* p_1/p_2 *represents the concatenation of* p_1 *and* p_2*,* $p_1[p_2]$ *represents a path* p_1 *with a* predicate p_2 *and finally* $\cap$*,* $\cup$ *and* $-$ *represent the set intersection, set union and set difference.*

[1] A strict total order is a binary relation that is irreflexive, transitive and total.

All remaining axes in XPath can be straightforwardly defined in terms of the given axes: (ancestor) $\uparrow^+ \equiv \uparrow/\uparrow^*$, (descendant) $\downarrow^+ \equiv \downarrow/\downarrow^*$, (preceding) $\twoheadleftarrow \equiv \uparrow^*/\dot{\twoheadleftarrow}/\downarrow^*$, (following) $\twoheadrightarrow \equiv \uparrow^*/\dot{\twoheadrightarrow}/\downarrow^*$. Also boolean expressions in predicates can be readily simulated: $p_1 \wedge p_2 \equiv (p_1/\Uparrow) \cap (p_2/\Uparrow)$, $p_1 \vee p_2 \equiv (p_1/\Uparrow) \cup (p_2/\Uparrow)$ and $\neg p_1 \equiv \Uparrow - (p_1/\Uparrow)$.

Based on [15] and [7] and similar to [2] we define the semantics of these expressions as follows:

Definition 3 (XPath Semantics). *Given an XML tree $T = (N, \lhd, r, \lambda, \prec)$ we define the* semantics of a path expression *p, $[\![p]\!]_T \subseteq N \times N$, such that $(n, n') \in [\![p]\!]_T$ iff one of the following applies: (1) if $p = \epsilon$ then $n = n'$, (2) if $p = \uparrow$ then $n' \lhd n$, (3) if $p = \downarrow$ then $n \lhd n'$, (4) if $p = \uparrow^*$ then $n' \lhd^* n$, (5) if $p = \downarrow^*$ then $n \lhd^* n'$, (6) if $p = \dot{\twoheadleftarrow}$ then $n' \prec n$ and there is an n'' such that $n'' \lhd n$ and $n'' \lhd n'$, (7) if $p = \dot{\twoheadrightarrow}$ then $n \prec n'$ and there is an n'' such that $n'' \lhd n$ and $n'' \lhd n'$, (8) if $p = \Uparrow$ then $n' = r$, (9) if $p = t \in \Sigma$ then $n = n'$ and $\lambda(n) = t$, (10) if $p = p_1/p_2$ then there is an n'' such that $(n, n'') \in [\![p_1]\!]_T$ and $(n'', n') \in [\![p_2]\!]_T$, (11) if $p = p_1[p_2]$ then $(n, n') \in [\![p_1]\!]_T$ and there is an n'' such that $(n', n'') \in [\![p_2]\!]_T$, (12) if $p = p_1 \cap p_2$ then $(n, n') \in [\![p_1]\!]_T$ and $(n, n') \in [\![p_2]\!]_T$, (13) if $p = p_1 \cup p_2$ then $(n, n') \in [\![p_1]\!]_T$ or $(n, n') \in [\![p_2]\!]_T$, and (14) if $p = p_1 - p_2$ then $(n, n') \in [\![p_1]\!]_T$ and $(n, n') \notin [\![p_2]\!]_T$.*

Remark 1. The tag names steps of the form $t \in \Sigma$ behave as if they follow the *self* axis. This means that `a/b` corresponds to the conventional XPath expression `self::a/self::b` and *not* to the expression `child::a/child::b` as is the case for the so-called abbreviated XPath syntax. Consequently the XPath expression `child::a/ancestor::b` can be represented in our syntax as $\downarrow$`/a/`$\uparrow^+$`/b` .

Fragments of P are denoted as $\mathcal{P}_V$ where V is a subset of $\{\Uparrow, [\,], \cap, \cup, -\}$. In $\mathcal{P}$ only expressions that consist of the axes, Σ and P/P are allowed. With the subscripts $\Uparrow$, $[\,]$, $\cap$, $\cup$ and $-$ also expressions of the form $\Uparrow$, $P[P]$, $P \cap P$, $P \cup P$ and $P - P$ are allowed, respectively.

Finally, we define what it means for an XPath expression to be satisfiable.

Definition 4 (XPath Satisfiability). *An XPath expression p is called* satisfiable *if there is an XML tree T such that $[\![p]\!]_T$ is not empty.*

Example 1. Given two distinct tag names a and b in Σ the following expressions are *not* satisfiable: a/b, $a[b]$, $a/\downarrow/\uparrow/b$, $\Uparrow/\dot{\twoheadleftarrow}$ and $a/\downarrow/\dot{\twoheadrightarrow}/\uparrow/b$.

3 Tree Description Graphs

Before we discuss the problem of deciding satisfiability of XPath expressions we consider the same problem for a related notion called *partial tree descriptions* which has been studied in computational linguistics [14, 5, 10, 4]. A partial tree description can be informally described as a formula in EFO (Existential First Order Logic) that quantifies over the nodes in the tree and uses the binary predicates $=$, $\lhd$, $\lhd^+$, $\lhd^*$ and $\prec$ and a special constant r (the root). Such formulas can be used to describe various properties of ordered trees and also to query such trees. For our purposes we will consider only formulas that have the conjunction as their only logical operation and extend them with unary predicates for each tag name $t \in \Sigma$. This leads to the notion of *tree description* graph.

Definition 5 (Tree Description Graph). *A* tree description graph *(TDG) is a tuple $D = (V, v_r, \Phi)$ with V a finite set of variables, v_r a special element in V that represents the root and Φ a set of atoms of the following forms: $t(v_1)$ with $t \in \Sigma$ denoting that v_1 is labelled with t, $v_1 = v_2$, $v_1 \lhd v_2$, $v_1 \lhd^* v_2$, $v_1 \lhd^+ v_2$, $v_1 \prec v_2$ with $v_1, v_2 \in V$.*

Example 2. An example of a TDG is $D = (V, v_r, \Phi)$ with a set of variables $V = \{v_r, v_1, \ldots, v_5\}$ and atoms $\Phi = \{v_r \lhd^* v_1,\ v_1 \lhd^+ v_2,\ v_1 \lhd^* v_3,\ v_1 = v_4,\ v_3 \lhd v_4,\ v_4 \prec v_5,\ v_2 = v_5,\ b(v_2),\ a(v_4)\}$. This tree description graph is shown in Fig. 1 where the $=$ predicate is indicated with double lines and the $\prec$ predicate is indicated with a dotted line.

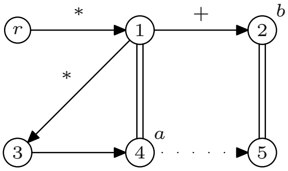


Fig. 1. A tree description graph.

Tree description graphs can be seen as a generalization of *tree patterns* [12] that allows more axes. Like for XPath expressions we can also define a notion of satisfiability for TDGs.

Definition 6 (TDG Satisfiability). *Given a TDG $D = (V, v_r, \Phi)$ a* model *for D is a tuple $M = (T, I)$ with an XML tree $T = (N, \lhd, r, \lambda, \prec)$ and an interpretation $I : V \to N$ such that $I(v_r) = r$ and I makes all atoms in Φ satisfied for T. A TDG is called* satisfiable *if there is a model for it.*

The TDG in Fig. 1 is not satisfiable because in a model (T, I) it would both have to hold that $I(v_1) = I(v_4)$ and $I(v_1) \lhd^+ I(v_4)$, which is not possible.

Similar to [12] we show that when reasoning about a certain property of patterns we only need to consider a limited set of models; for deciding satisfiability we only need to consider models that have the same size as the TDG.

Lemma 1. *If a tree description graph $D = (V, v_r, \Phi)$ is satisfiable then there is a model (T, I) for D with at most $|V|$ nodes in T.*

Proof. We show that if for the TDG $D = (V, v_r, \Phi)$ there is a model $M_1 = (T_1, I_1)$ with XML tree $T_1 = (N_1, \lhd_1, r_1, \lambda_1, \prec_1)$ and $|N_1| > |V|$ then there is a model for D with one node less than M_1. By induction upon $|N_1|$ it then follows that the lemma holds.

The smaller model is $M_2 = (T_2, I_1)$ where T_2 is constructed as follows. We choose an arbitrary node n in N_1 that is not in the image of I_1, i.e., there is no $v \in V$ such that $I_1(v) = n$. Such a node exists since $|N_1| > |V|$. We then construct T_2 by (1) removing n from the tree and (2) making the children of n now the children of the parent of n. Note that n will always have a parent since the only node that has no parent is the root and the root is always in the image of I_1. More formally, we define $T_2 = (N_2, \lhd_2, r_2, \lambda_2, \prec_2)$ such that (1) $N_2 = N_1 - \{n\}$, (2) $\lhd_2 = (\lhd_1 \cap (N_2 \times N_2)) \cup \{(n', n'') | n' \lhd_1 n \wedge n \lhd_1 n''\}$, (3) $r_2 = r_1$, (4) $\lambda_2 = \lambda_1|_{N_2}$ where $\lambda_1|_{N_2}$ is the restriction of the function λ_1 to the domain N_2, and (5) $\prec_2 = \prec_1 \cap (N_2 \times N_2)$. Then it can be shown that T_2 is indeed an XML tree and (T_2, I_1) is a model for D:

T_2 is an XML Tree : By its construction T_2 defines a finite labelled tree. What remains to be shown is that $\prec$ defines a pre-order tree-walk. The condition PTW1 holds for T_2 because if $n_1 \lhd_2 n_2$ then by the construction of $\lhd_2$ it follows that $n_1 \lhd_1^+ n_2$ and so $n_1 \prec_1 n_2$ which implies $n_1 \prec_2 n_2$. The condition PTW2 also holds for T_2, which can be shown as follows. Assume that $n_3 \lhd_2 n_1$, $n_3 \lhd_2 n_2$, $n_1 \prec_2 n_2$ and $n_1 \lhd_2^+ n_4$. By the construction of T_2 it will hold that $n_1 \prec_1 n_2$ and $n_1 \lhd_1^+ n_4$. It will also hold for the removed node n that either (a) $n_3 \lhd_1 n \lhd_1 n_1$, or (b) $n_3 \lhd_1 n \lhd_1 n_2$, or (c) $n_3 \lhd_1 n_1$ and $n_3 \lhd_1 n_2$. In all cases we can derive that $n_4 \prec_1 n_2$ as follows. In case (a) it holds that in T_1 the node n was a smaller sibling of n_2 and therefore $n_4 \prec_1 n_2$. In case (b) it holds that in T_1 the node n was a larger sibling of n_1 and therefore $n_4 \prec_1 n$. By TW1 it also holds that $n \prec_1 n_2$, so it follows that $n_4 \prec_1 n_2$. In case (c) it directly follows by TW2 for T_1 that $n_4 \prec_1 n_2$. Summarizing we now know that in all cases $n_4 \prec_1 n_2$ and since both nodes are in N_2 it follows that $n_4 \prec_2 n_2$.

(T_2, I_1) is a model for D: Because $r_1 = r_2$ and λ_2 and $\prec_2$ are restrictions of λ_1 and $\prec_1$, respectively, to N_2 it follows that $I_1(v_r) = r_2$ and I_1 makes the t atoms and the $\prec$ atoms satisfied for T_2. Because $\lhd_2^+$ and $\lhd_2^*$ are equal to restrictions of $\lhd_1^+$ and $\lhd_1^*$ to N_2 it also holds for these atoms that I_1 makes them satisfied for T_2. Finally, because $\lhd_2$ is a superset of the restriction of $\lhd_1$ to N_2 it also holds that I_1 satisfies the $\lhd$ atoms for T_2.

The previous observation then leads to the following theorem.

Theorem 1. *Deciding satisfiability of a tree description graph is in NP.*

Proof. It is easy to see that there is a model for a TDG $D = (V, v_r, \Phi)$ iff there is a model for $D' = (V', v_r, \Phi)$ where V' is the subset of V that is mentioned in Φ plus v_r. By Lemma 1 it holds that D' is satisfiable iff there is a model for D' with at most $|V'|$ nodes. If the size of the representation of D' is n then there can be no more than $2n$ variables in V'. It follows that D is satisfiable iff there is a model for D with a representation size of $\mathcal{O}(n^2)$. It follows that a non-deterministic algorithm can guess an XML tree T and an interpretation I in a polynomial number of steps. Since it can also be checked in polynomial time that (T, I) is a model of D' it follows that there is an NP algorithm that decides whether there is a model for D.

4 TDGs and XPath

Both TDGs and XPath expressions can be used to define binary relations over the nodes of a given XML tree. For example, the binary relation defined by the path expression $(\downarrow^*/a/\uparrow/b/\uparrow^*) \cap (\downarrow^+/c/\dot{\twoheadrightarrow})$ is also defined by the TDG in Fig. 2 where the begin and end variable are indicated by a bold incoming and leaving arrow, respectively.

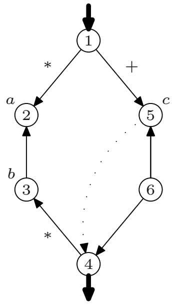


Fig. 2. A TDG representing an XPath expression.

In general the translation of an XPath expression in the fragment $\mathcal{P}_{\Uparrow,[\,],\cap}$ to a TDG is defined as follows.

Definition 7 (Atom Set). *Given an XPath expression $p \in \mathcal{P}_{\Uparrow,[\,],\cap}$, a begin variable v and an end variable v' the* atom set for p from v to v', *denoted as $\Phi_{v,v'}(p)$, is defined as follows:*

$$\begin{aligned}
\Phi_{v,v'}(\epsilon) &= \{v = v'\} \\
\Phi_{v,v'}(\uparrow) &= \{v' \lhd v\} \\
\Phi_{v,v'}(\downarrow) &= \{v \lhd v'\} \\
\Phi_{v,v'}(\uparrow^*) &= \{v' \lhd^* v\} \\
\Phi_{v,v'}(\downarrow^*) &= \{v \lhd^* v'\} \\
\Phi_{v,v'}(\dot{\twoheadleftarrow}) &= \{v' \prec v, v'' \lhd v, v'' \lhd v'\} \textit{with } v'' \textit{ a fresh variable} \\
\Phi_{v,v'}(\dot{\twoheadrightarrow}) &= \{v \prec v', v'' \lhd v, v'' \lhd v'\} \textit{with } v'' \textit{ a fresh variable} \\
\Phi_{v,v'}(\Uparrow) &= \{v' = v_r\} \\
\Phi_{v,v'}(t) &= \{v = v', t(v')\} \\
\Phi_{v,v'}(p_1/p_2) &= \Phi_{v,v''}(p_1) \cup \Phi_{v'',v'}(p_2) \textit{with } v'' \textit{ a fresh variable} \\
\Phi_{v,v'}(p_1[p_2]) &= \Phi_{v,v'}(p_1) \cup \Phi_{v',v''}(p_2) \textit{with } v'' \textit{ a fresh variable} \\
\Phi_{v,v'}(p_1 \cap p_2) &= \Phi_{v,v'}(p_1) \cup \Phi_{v,v'}(p_2)
\end{aligned}$$

The correctness of this translation is established by the following theorem.

Theorem 2. *Given a path expression $p \in \mathcal{P}_{\Uparrow,[\,],\cap}$ and if V is the set of all variables in $\Phi_{v,v'}(p)$ plus v_r then for every XML tree T and nodes n, n' in T it holds that there is a model (T, I) for $(V, v_r, \Phi_{v,v'}(p))$ with $I(v) = n$ and $I(v') = n'$ iff $(n, n') \in [\![p]\!]_T$.*

Proof. (Sketch) This can be shown with induction upon the structure of p and follows straightforwardly from the given semantics of XPath expressions.

Whether each TDG can be translated to an equivalent expression in the fragment $\mathcal{P}_{\Uparrow,[\,],\cap}$ is still an open problem.

It follows from this translation that deciding satisfiability of path expressions in $\mathcal{P}_{\Uparrow,[\,],\cap}$ is in NP.

Theorem 3. *Deciding satisfiability of path expressions in $\mathcal{P}_{\Uparrow,[\,],\cap}$ is in NP.*

Proof. Satisfiability of the path expression p can be decided by translating it to the corresponding TDG and deciding if this is satisfiable. The translation can be done in PTIME and by Lemma 1 we can decide satisfiability of TDGs in NP.

5 String Matching Problems

In order to show the hardness of deciding satisfiability for certain XPath fragments we will show that the following string matching problem can be reduced to these problems.

Definition 8 (Bounded Multiple String Matching Problem). *Given a finite set of patterns A, which are strings over $\{0, 1, *\}$, is there a string over $\{0, 1\}$ whose size is equal to the size of the largest pattern in A and in which all patterns in A can be matched with $*$ as a wildcard for one symbol?*

In the following we will also refer to this problem as the BMS problem.

Theorem 4. *Deciding the BMS problem is NP-complete.*

Proof. It is easy to see that such a string can be guessed and verified in non-deterministic polynomial time

To prove NP-hardness we show that there is a polynomial reduction from 3SAT [9] to this problem. The reduction consists of a mapping of a CNF formula to a set of formulas such that there is a string into which all these patterns can be matched and that is as large as the largest pattern iff this string encodes a truth assignment for the formula.

To demonstrate the principle we will first show how the formula $\varphi = C_1 \wedge C_2$ with $C_1 = X_1 \vee \neg X_2 \vee X_3$ and $C_2 = \neg X_1 \vee X_2 \vee \neg X_4$ is translated. The encoding of the truth assignment is illustrated by the first pattern in Fig. 3, which is called a_{pre}.

$$\begin{array}{rcccccccccccccccc}
 & & & & X_1^{(1)} & & & X_2^{(1)} & & & X_3^{(1)} & & & X_4^{(1)} & & \\
a_{pre} = & \texttt{101010} & \texttt{**} & \texttt{**} & \underline{\texttt{**}} & \texttt{**} & \texttt{**} & \underline{\texttt{**}} & \texttt{**} & \texttt{**} & \underline{\texttt{**}} & \texttt{**} & \texttt{**} & \underline{\texttt{**}} & \texttt{**} & \texttt{**} \\
 & 1 & 7 & & 11 & & & 17 & & & 23 & & & 29 & & 34 \\
 & & & & X_1^{(2)} & & & X_2^{(2)} & & & X_3^{(2)} & & & X_4^{(2)} & & \\
 & & \texttt{**} & \texttt{**} & \underline{\texttt{**}} & \texttt{**} & \texttt{**} & \underline{\texttt{**}} & \texttt{**} & \texttt{**} & \underline{\texttt{**}} & \texttt{**} & \texttt{**} & \underline{\texttt{**}} & \texttt{**} & \texttt{**} \\
 & & 35 & & 39 & & & 45 & & & 51 & & & 57 & & 62
\end{array}$$

$$\begin{array}{rcccccccccccccccc}
a^1_{C_1,X_1} = & \texttt{******} & \texttt{**} & \texttt{**} & \texttt{1*} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} \\
 & & \texttt{**} & \texttt{**} & \texttt{1*} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{*} \\
a^0_{C_1,X_1} = & \texttt{******} & \texttt{**} & \texttt{**} & \texttt{0*} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} \\
 & & \texttt{**} & \texttt{**} & \texttt{0*} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{*} \\
\ldots \quad = & \ldots & & & & & & & & & & & & & & \\
a^1_{C_1,X_4} = & \texttt{******} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{1*} & \texttt{**} & \texttt{**} \\
 & & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{1*} & \texttt{**} & \texttt{*} \\
a^0_{C_1,X_4} = & \texttt{******} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{0*} & \texttt{**} & \texttt{**} \\
 & & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{**} & \texttt{0*} & \texttt{**} & \texttt{*}
\end{array}$$

$$\begin{array}{rcccccccc}
 & & X_1^{(1)} & X_2^{(1)} & X_3^{(1)} & X_4^{(1)} & & \\
a_{C_1} = & \texttt{1*****} & \underline{\texttt{10 ** **}} & \underline{\texttt{** 01 **}} & \underline{\texttt{** ** 10}} & \underline{\texttt{** ** **}} & \texttt{**} & \texttt{**} \\
 & 1 & 7 & 13 & 19 & 25 & 31 & 34 \\
 & & X_1^{(2)} & X_2^{(2)} & X_3^{(2)} & X_4^{(2)} & & \\
 & & \texttt{** ** **} & \texttt{** ** **} & \texttt{** ** **} & \texttt{** ** **} & & \\
 & & 35 & 41 & 47 & 53 \quad 58 & & \\
a_{C_2} = & \texttt{1*****} & \texttt{** ** **} & \texttt{** ** **} & \texttt{** ** **} & \texttt{** ** **} & \texttt{**} & \texttt{**} \\
 & & \underline{\texttt{01 ** **}} & \underline{\texttt{** 10 **}} & \underline{\texttt{** ** **}} & \underline{\texttt{** ** 01}} & &
\end{array}$$

Fig. 3. A set of patterns for the formula $C_1 \wedge C_2$ with $C_1 = X_1 \vee \neg X_2 \vee X_3$ and $C_2 = \neg X_1 \vee X_2 \vee \neg X_4$.

The pattern a_{pre} will be the longest pattern and therefore defines the length of the string, in this case 62 characters. It also enforces that the string begins with 101010. The underlined positions marked by $X_j^{(i)}$ are intended to encode a truth assignment for variable X_j in clause C_i; the pair 10 denotes *true* and 01 denotes *false*.

Because in the chosen encoding every clause has a separate truth assignment we introduce the patterns $a^1_{C_1,X_1}, a^0_{C_1,X_1}, \ldots, a^1_{C_1,X_4}, a^0_{C_1,X_4}$. These patterns are one character shorter than a_{pre} and each pattern $a^1_{C_1,X_j}$ ($a^0_{C_1,X_j}$) contains two 1s (0s); one at the first position of the pair marked with $X_j^{(1)}$ and the other at $X_j^{(2)}$. Because of their length these patterns can only be embedded in the string in two ways; starting from the first or from the second position in the string. It is then easy to see that the patterns $a^1_{C_1,X_1}$ and $a^0_{C_1,X_1}$ enforce that the positions marked by $X_1^{(1)}$ and $X_1^{(2)}$ contain either both 10 or both 01. If we introduce such a pattern for each variable then we can ensure that all truth assignment for all clauses are the same.

Finally, we introduce the patterns a_{C_1} and a_{C_2} to ensure that the clauses C_1 and C_2, respectively, are satisfied. First note that these patterns start with 1 and their length is 4 less than the string. Since a_{pre} enforces that the string starts with 101010 it follows that these patterns can only be matched in three ways; starting from the first, third or fifth position. The construction of a_{C_1} is now as follows. For each variable X_j there is a region of six characters underlined and marked with $X_j^{(1)}$. In this region we set the kth pair to 10 if the variable appears in the kth position in the clause, and to 01 if the negation of the variable appears at that position. So for the clause $C_1 = X_1 \vee \neg X_2 \vee X_3$ the first pair of X_1's region is set to 10, the second pair in X_2's region is set to 01 and the third pair in X_3's regions is set to 10.

Now consider what happens with each way that this pattern can be matched in the string.

1. If it is matched from the *first* position in the string then the pair 10 at positions [23,24] is mapped to the same positions in the string which are marked as $X_3^{(1)}$ in a_{pre}, and all other 10 and 01 pairs are mapped to unmarked positions
2. If the pattern a_{C_1} is matched from the *third* position then the pair 01 at positions [15,16] is mapped to the positions [17,18] in the string which are marked as $X_2^{(1)}$ in a_{pre}, and all other 10 and 01 pairs are mapped to unmarked positions.
3. Finally, if the pattern a_{C_1} is matched from the *fifth* position then the pair 10 at positions [7,8] is mapped to the positions [11,12] in the string which are marked as $X_1^{(1)}$ in a_{pre}, and all other 10 and 01 pairs are mapped to unmarked positions.

Summarizing, if the pattern matches then the encoded truth assignment for $X_1^{(1)}, \ldots, X_4^{(1)}$ will make at least one literal in the clause C_1 true. In a similar fashion the pattern a_{C_2} ensures that the encoded truth assignment for $X_1^{(2)}, \ldots, X_4^{(2)}$ will make at least one literal in C_2 true.

We can now summarize the meaning of the patterns in Fig. 3 as follows. The pattern a_{pre} defines a preamble and the lengt of the string such that there is room for a separate truth assignment for each clause. The patterns $a^1_{C_1,X_1}, a^0_{C_1,X_1}, \ldots, a^1_{C_1,X_4}, a^0_{C_1,X_4}$ ensure that all the truth assignments for the different clauses are in fact the same truth assignment. Finally, the patterns a_{C_1} and a_{C_2} ensure that the truth assignment for C_1 makes C_1 satisfied and the truth assignment for C_2 makes C_2 satisfied. It then follows that a string of the same length as a_{pre} into which all these patterns match, defines a truth assignment that makes φ satisfied.

On the other hand, if there is a truth assignment that makes φ satisfied then we can construct a string of the same length as a_{pre} such that all patterns match into this string as follows. As required by a_{pre} we let the string start with 101010 and we encode the truth assignment in the string in the positions that are marked for a_{pre}. Since we assign the same truth assignment for all clauses the patterns $a^1_{C_1,X_1}, a^0_{C_1,X_1}, \ldots, a^1_{C_1,X_4}, a^0_{C_1,X_4}$ will all match. Finally we can make sure that a_{C_1} and a_{C_2} match into the string by choosing for each clause one literal that is made true by the assignment and mapping it to the position in a_{pre} that is marked for that literal. Since all other 10 and 01 in the pattern will then be mapped to unmarked positions it follows that we can dan fill these positions in the string such that the pattern indeed matches. For example, if the truth assignment maps X_2 to *false* then we might map a_{C_1} into the string such that the positions [15,16] are mapped to [17,18] in the string, and therefore positions [7,8] and [23,24] are mapped to [9,10] and [25,26], respectively, and therefore these positions in the string should contain the pairs at [7,8] and [23,24] in a_{C_1}, i.e., in both cases `10`.

Summarizing, we have shown that the BSM problem defined by the patterns in Fig. 3 is satisfiable iff the formula φ is satisfiable. This concludes the example and we will now proceed with a description of the reduction in general.

Let us consider a formula $\varphi = C_1 \wedge \ldots \wedge C_m$ with $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$ with $l_{i,k} = X_j$ or $l_{i,k} = \neg X_j$ where $X_1, \ldots, X_n$ are the variables in φ.

The first pattern defines the preamble and the length of the string:

$$a_{pre} = \mathtt{101010}\ *^{(4+6n)m}$$

Note that the total length of this pattern is $6+(4+6n)m$ and that the assignment of variable X_j for clause C_i can be found in the pair that starts at position $7+(4+6n)(i-1)+4+6(j-1)$ in the string.

We then proceed with constructing the patterns that ensure that the truth assignment for variable X_j in C_i is equal to that in C_{i+1}:

$$a^1_{C_i,X_j} = * * * * * ** *^{(4+6n)(i-1)} *^{4+6(j-1)}\ \mathtt{1}*^{(3+6n)}\mathtt{1}*^{(6n-2)-6(j-1)}\ *^{(4+6n)(m-i-1)}$$
$$a^0_{C_i,X_j} = * * * * * ** *^{(4+6n)(i-1)}\ *^{4+6(j-1)}\mathtt{0}*^{(3+6n)}\mathtt{0}*^{(6n-2)-6(j-1)}\ *^{(4+6n)(m-i-1)}$$

Because the length of these two patterns is $6+(4+6n)m-1$ there are only two ways in which it can be matched with a string of length $6+(4+6n)m$.

Finally we define the patterns that ensure that the encoded truth assignment make a certain clause satisfied. For this purpose we define $a[C_i, X_j]$ as equal to `******` except that the kth pair is equal `10` if $l_{i,k} = X_j$ and equal to `01` if $l_{i,k} = \neg X_j$. We then construct the patterns that ensure that the clause C_i is made satisfied by the truth assignment for C_i as follows.

$$a_{C_i} = \mathtt{1} * * * ** *^{(4+6n)(i-1)}\ a[C_i, X_1] \ldots a[C_i, X_n]\ *^{(4+6n)(m-i)}$$

Note that a_{C_i} is padded with `*`s to a length of $2+(4+6n)m$ to ensure that it can only be matched to the 1st, 3rd and 5th position in the string that has the length of a_{pre} and into which a_{pre} matches.

The total set of patterns for φ is now defined as

$$A_\varphi = \{a_{pre}\} \cup \{a^0_{C_i,X_j}, a^1_{C_i,X_j} \mid 1 \leq i < m, 1 \leq j \leq n\} \cup \{a_{C_i} \mid 1 \leq i \leq m\}$$

It can be shown that all the patterns in A_φ are bounded polynomially in n and m:

$$|a_{pre}| = 6 + (4+6n)m$$
$$|a^b_{C_i,X_j}| = 6 + (4+6n)m - 1$$
$$|a_{C_i}| = 2 + (4+6n)m$$

It follows that they are polynomially bounded by the size of φ and because there are $1 + 2(m-1)n + m$ patterns in A_φ it also holds that the representation of this set is bounded polynomially in the size of φ. Consequently it is easy to see that A_φ can be generated from φ in polynomial time.

What remains to be shown is that A_φ is satisfiable iff φ is satisfiable.

It is easy to see that if a string x satisfies A_φ then we can read a truth assignment that satisfies from the position for the truth assignment for X_j in any C_i. Because of the $a^b_{C_i,X_j}$ patterns these assignment will be the same for any clause C_i and because of the a_{C_i} patterns all clauses in φ will be satisfied.

If there is a truth assignment that satisfies φ then we can construct x as follows. We start the string with `101010` and fill in x the positions for X_j for each clause C_i as prescribed by x. This ensures that the a_{pre} pattern and the $a^b_{C_i,X_j}$ patterns are satisfied. Next, we pick in each clause one of the three literals that is satisfied by the truth assignment and map the a_{C_i} patterns accordingly to x and set the `1`s and `0`s that are required by them. Finally, the remaining positions in x can be filled with arbitrary `1`s and `0`s.

6 Lower-Bound Results

We now proceed with discussing the hardness of deciding satisfiability of fragments of XPath. The first hardness result concerns $\mathcal{P}_\cap$, i.e., the fragment that allows only expressions that consist of the axes and expressions of the form Σ, P/P and $P \cap P$.

Theorem 5. *Deciding satisfiability of path expressions in $\mathcal{P}_\cap$ is NP-hard.*

Proof. We show that the BMS problem can be reduced to this problem. We assume that the set of patterns is $\{a_0, \ldots, a_n\}$ and that a_0 is the longest pattern. The pattern a_0 is translated to a path p_0 by translating a 0 to $\downarrow/a$, 1 to $\downarrow/b$ and $*$ to just $\downarrow$. For example, "*0*0*1" is translated to $\downarrow/\downarrow/a/\downarrow/\downarrow/a/\downarrow/\downarrow/b$. The other patterns a_i are translated to p_i in the same way but with an extra $\downarrow^*$ step before and after it. So, for example, "10" is translated to $\downarrow^*/\downarrow/b/\downarrow/a/\downarrow^*$. Finally we take the intersection of all these paths: $p_0 \cap p_1 \cap \ldots \cap p_n$.

It is easy to see that if there is an XML tree T and a pair (n, n') in the semantics of this path under T then labels of the nodes in the path from n to n' represent a string into which all patterns match if we replace a and b with 1 and 0, respectively.

Conversely, if there is a string into which all pattern can be matched then we can construct an XML tree that consists of a simple path that is labelled with the labels that correspond with the characters in the string, for which the semantics of the path expression will contain at least (n, n') with n and n' the begin and end node of this path, respectively.

Remark 2. In the proof we only need the forward axes $\downarrow$ and $\downarrow^*$ and the ordering of the trees is not used.

Theorem 6. *Deciding satisfiability of tree description graphs is NP-hard.*

Proof. This follows from the straightforward translation of path expressions in $\mathcal{P}_{\Uparrow,[\,],\cap}$ as given in Def. 7 and Th. 5.

Theorem 7. *Deciding satisfiability of path expressions in $\mathcal{P}_{-}$ is NP-hard.*

Proof. This proof proceeds similar to the one of Th. 5 except that the path $p_0 \cap p_1 \cap \ldots \cap p_n$ is simulated with $p_0 - (p_0 - p_1) - (p_0 - p_2) - \ldots - (p_0 - p_n)$.

Theorem 8. *Deciding satisfiability of path expressions in $\mathcal{P}_{[\,],\cup}$ is NP-hard.*

Proof. We show this by reducing the problem SAT [9]. We construct for every CNF formula $\varphi = C_1 \wedge \ldots \wedge C_m$ a path p_φ as follows. Let the variables in φ be $X_1, \ldots, X_n$. For every literal l we define a path p_l such that p_{X_i} is a path of $n+1$ steps of the form $\uparrow$ except step $i+1$ which is of the form a, and $p_{\neg X_i}$ is the same except that step $i+1$ is of the form b. For example, for $n = 3$:

$$\begin{aligned} p_{X_2} &= \uparrow/\uparrow/a/\uparrow \\ p_{\neg X_3} &= \uparrow/\uparrow/\uparrow/b \end{aligned}$$

A clause $l_1 \vee \ldots \vee l_p$ is straightforwardly mapped to $p_{l_1} \cup \ldots \cup p_{l_p}$. For example $p_{X_2 \vee \neg X_3}$ is

$$(\uparrow/\uparrow/a/\uparrow) \cup (\uparrow/\uparrow/\uparrow/b)$$

Finally, the formula $C_1 \wedge \ldots \wedge C_m$ is mapped to $\epsilon[p_{C_1}] \ldots [p_{C_m}]$.

It is easy to see that p_φ is satisfiable iff φ is satisfiable. Moreover, if k is the length of φ then $m \leq k$, $n \leq k$ and k will also be the upper-bound for the number of literals per clause, and therefore the size of p_φ will be in $\mathcal{O}(k^3)$.

Theorem 9. *Deciding satisfiability for $\mathcal{P}_{\Uparrow,[\,]}$ is NP-hard.*

Proof. Similar to the proof of Th. 5 we show that the BMS problem can be reduced to this problem. We assume that the set of patterns is $\{a_0, \ldots, a_n\}$ and that a_0 is the longest pattern. The pattern a_0 is translated to a path p_0 by starting with a $\Uparrow$ followed by translating a 0 to $\downarrow/a$, 1 to $\downarrow/b$ and $*$ to just $\downarrow$. For example, "*0*0*1" is translated to $\Uparrow/\downarrow/\downarrow/a/\downarrow/\downarrow/a/\downarrow/\downarrow/b$. The other patterns a_i are translated to p_i in a similar fashion but here we start with $\uparrow^*$ and then translate the pattern *in reverse* and with the $\uparrow$ axis in stead of the $\downarrow$ axis. For example, "11*0" is translated to $\uparrow^*/\uparrow/b/\uparrow/\uparrow/a/\uparrow/a$. Finally we construct from these paths the following path: $p_0[p_1][p_2] \ldots [p_n]$.

As in the proof in Th. 5 it holds for this path expression that if there is a pair in its semantics for a certain XML tree then the labels of the nodes in the path between those nodes corresponds to a string that satisfies the original BSM problem. Conversely, if there is a string that satisfies the BSM problem then a path that is labelled correspondingly and starts from the root will constitute an XML tree that satisfies the path expression.

Remark 3. Unlike the proof of Th. 5 this one requires forward axes ($\downarrow$) and backward axes ($\uparrow^*$ and $\uparrow$), but still does not use axes based on document order.

7 Upper-Bound Results

In this section we discuss some upper-bounds for XPath fragments. For the very large fragment $\mathcal{P}_{\Uparrow,[\,],\cap,\cup}$ that allows everything except the set difference, it can be shown that deciding satisfiability is in NP.

Theorem 10. *Deciding satisfiability of path expressions in $\mathcal{P}_{\Uparrow,[\,],\cap,\cup}$ is in NP.*

Proof. The algorithm starts with guessing non-deterministically for every subexpression of the form $p_1 \cup p_2$ if it replaces it with just p_1 or p_2 and for the resulting $\mathcal{P}_{\Uparrow,[\,],\cap}$ expression it decides with the algorithm of Th. 3 if the resulting $\mathcal{P}_{\cap}$ expression is satisfiable.

Remark 4. At the moment we don't have an upper bound for $\mathcal{P}_{-}$ and it is not even known if it decidable.

Theorem 11. *Deciding satisfiability for $\mathcal{P}_{[\,]}$ is in PTIME.*

Proof. (Sketch) We start with using Def. 7 to transform the path expression to a TDG. The result will be essentially a tree except for small cycles of three nodes to simulate the $\dot{\twoheadrightarrow}$ and $\dot{\twoheadleftarrow}$ axes.

We then apply the following rules to this graph until they can be applied no more:

1. If there is an atom $v_i = v_j$ then it is removed and all occurrences of v_i are replaced by v_j, i.e., the nodes v_i and v_j are merged.
2. If there are two distinct atoms $v_i \lhd v_j$ and $v_k \lhd v_j$ then all occurrences of v_i are replaced by v_k, i.e., the nodes v_i and v_k are merged.

This can create more cycles but it will always hold for each undirected cycle, i.e., a cycle that ignores the direction of the edges, that (p1) it contains only $\lhd$ and $\prec$ edges and (p2) is not a directed cycle, because these properties hold for the initial TDG and are preserved by the rules. Another property for which this holds is that (p3) if there is an $\prec$ edge between two nodes then there are two $\lhd$ edges that define a common parent. Because the rules are applied exhaustively it will also hold in the result that (p4) there are no two distinct atoms v_i and v_k for which there is a node v_j such that $v_i \lhd v_j$ and $v_k \lhd v_j$.

Finally, we check if there is a conflict, i.e., there are two atoms $a(v_i)$ and $b(v_i)$ with $a \neq b$. If so then this TDG is not satisfiable and because all the applied rules maintain satisfiability also the original TDG and consequently also the original path expression is not satisfiable.

If there is no such conflict then we can construct a satisfying XML tree from the obtained TDG as follows. We divide the variables into clusters which are maximal sets of variables that are directly or indirectly connected by $\lhd$ atoms. Note that because of properties p2 and p4 the $\lhd$ atoms define a tree over the variables in each cluster and because of p2 and p3 it is possible to complete the $\prec$ relationship for this tree to a strict total order that satisfies PTW1 and PTW2. Moreover, because of the properties p1 and p3 there can only be $\lhd^*$

and $\lhd^+$ edges between variables in different clusters and these edges will never define directed or undirected cycles over these clusters. Therefore we can sort the clusters topologically and connect the trees for each cluster by considering each cluster and its immediate successor in the topological sort (if there is one) and if there is an $\lhd^*$ or $\lhd^+$ edge from v_i in the first cluster to v_j in the second cluster then we add an $\lhd$ edge from v_i to the root of v_js cluster, if there is not an $\lhd^*$ or $\lhd^+$ edge between the clusters then we add an $\lhd$ edge between an arbitrary node in the first clusters with the root of the second cluster.

Finally, we have to complete the $\prec$ relationship, which was already completed for each cluster, for the complete tree. Since there are only $\lhd^+$ and $\lhd^*$ edges between the clusters and these define a tree over these clusters, it follows that we can complete the $\prec$ relationship to a strict total order over all the nodes that satisfies PTW1 and PTW2.

Theorem 12. *Deciding satisfiability for $\mathcal{P}_{\Uparrow}$ is in PTIME.*

Proof. (Sketch) This proof proceeds similar to the proof of Th. 11, but now we also merge v_i and v_r if there is an atom $v_i \lhd^* v_r$. Furthermore we also check for conflicts in the form of atoms $v_i \prec v_r$, $v_i \lhd v_r$ and $v_i \lhd^+ v_r$. Finally, we attempt to construct a satisfying XML tree in the same way except when there is an $\lhd^*$ or $\lhd^+$ edge that arrives in the cluster that contains the root variable v_r. Note that if we follow the procedure of the previous proof then this root node would become the child of another node, which is not allowed in an XML tree. Therefore we do here the following. Let the atom in question be $v_i \lhd^+ v_j$ or $v_i \lhd^* v_j$. Then we attempt to merge v_i and its ancestors (as defined by the $\lhd$ atoms) with an ancestor of v_j and its ancestors. If this is possible without a conflict between their tag names and without introducing a parent of the root then we merge them such that v_i is merged with the lowest possible ancestor of v_j in its cluster. This is repeated until the cluster with v_r has no more incoming $\lhd^*$ and $\lhd^+$ edges.

If by then we still have not found a conflict then we can proceed to construct the satisfying XML tree as in the previous proof by making sure that the cluster with v_r becomes the smallest cluster. If we do find a conflict then it is not possible to avoid it by merging v_i with a higher ancestor because that would only limit the possibilities more for subsequent merges. This is because it holds that the cluster with v_r has always just one incoming $\lhd^*$ or $\lhd^+$ edge unless the original path expression used $\Uparrow$ in other places then the beginning of the path. However, in the latter case we can decide satisfiability by splitting the path at the intermediate $\Uparrow$ step and deciding it separately for the two resulting path expressions.

8 Summary and Discussion

For tree description graphs the problem of deciding satisfiability was shown to be NP complete. This result is similar to that in [10] except that they require atoms of the form $v_0 : f(v_1, \ldots, v_n)$ that specify that v_0 is labelled with f and whose set of children is exactly $\{v_1, \ldots, v_n\}$. Our result shows that even with only unary atoms ($n = 1$) the problem is already NP hard.

For fragments of XPath the complexity results are given by the following table.

$\Uparrow$	$[\,]$	$\cup$	$\cap$	$-$	**Complexity**
●					PTIME
	●				PTIME
			●		NP-complete
●	●				NP-complete
	●	●			NP-complete
●	●	●	●		NP-complete
				●	NP-hard

Remaining open problems are finding a better lower bound and an upper bound for $\mathcal{P}_{-}$ and classifying the fragments $\mathcal{P}_{\cup}$ and $\mathcal{P}_{\Uparrow,\cup}$.

Another open problem is the relationship between $\mathcal{P}_{\Uparrow,[\,],\cap}$ and tree description graphs. As was shown by Th. 2 every path expression in this fragment can be translated to an equivalent TDG, but whether the converse holds is still unknown.

Finally, given the research that has been done on the containment problem for XPath expressions given a DTD [6, 13, 17] which limits itself mainly to XPath fragment with forward axes, and the results in this paper that seem to indicate that the satisfiability problem is sometimes simpeler, even if also reverse axes are allowed, it will be interesting to see what the complexity of the satisfiability problem is in the context of DTDs. Although some algorithms have been suggested such as in [11] this is still largely unknown.

Acknowledgement

I would like to thank the anonymous referees for their corrections, comments and suggestions for improvement.

References

1. Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *SIGMOD Conference*, 2001.
2. Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural properties of XPath fragments. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, volume 2572 of *LNCS*, pages 79–95. Springer, 2003.
3. Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML path language (XPath) 2.0. W3C Working Draft, 2002. `http://www.w3.org/TR/xpath20/`.
4. Manuel Bodirsky and Martin Kutz. Pure dominance constraints. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2002)*, 2002.

5. Tom Cornell. On determining the consistency of partial descriptions of trees. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 163–170. Morgan Kaufmann, 1994.
6. Alin Deutsch and Val Tannen. Containment and integrity constraints for XPath fragments. In *Proceedings of the 8th International Workshop on Knowledge Representation Meets Databases (KRDB 2001)*, 2001.
7. Denise Draper, Peter Fankhauser, Mary F. Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics, 2002. `http://www.w3.org/TR/xquery-semantics/`.
8. Mary F. Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 data model. W3C Working Draft, 2002. `http://www.w3.org/TR/xpath-datamodel/`.
9. Michael R. Garey and David S. Johnson. *Computers and Intractability – A guide to NP-completeness*. W. H. Freeman and Company, San Francisco, 1979.
10. Alexander Koller, Joachim Niehren, and Ralf Treinen. Dominance constraints: Algorithms and complexity. In *Proceedings of the Third Conference on Logical Aspects of Computational Linguistics*, volume 2014 of *Lecture Notes in Computer Science*, pages 106–125, Grenoble, 2001. Springer - Verlag.
11. April Kwong and Michael Gertz. Schema-based optimization of xpath expressions. Technical report, University of California at Davis, 2002.
12. Gerome Miklau and Dan Suciu. Containment and equivalence for an XPath fragment. In *Symposium on Principles of Database Systems*, pages 65–76, 2002.
13. Frank Neven and Thomas Schwentick. XPath containment in the presence of disjunction, DTDs and variables. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, volume 2572 of *LNCS*, pages 315–329. Springer, 2003.
14. James Rogers and K. Vijay-Shanker. Reasoning with descriptions of trees. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics*, pages 72–80, 1992.
15. Philip Wadler. Two semantics for XPath, 1999. `http://www.cs.bell-labs.com/who/wadler/topics/xml.html`.
16. Peter T. Wood. Minimising simple XPath expressions. In *Proceedings of the Fourth International Workshpo on the Web and Databases (WebDB 2001)*, pages 13–18, Santa Barbara, California, May 2001.
17. Peter T. Wood. Containment for XPath fragments under DTD constraints. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, volume 2572 of *LNCS*, pages 300–314. Springer, 2003.

Containment of Relational Queries with Annotation Propagation

Wang-Chiew Tan[*]

University of California, Santa Cruz
wctan@cs.ucsc.edu

Abstract. We study the problem of determining whether a query is contained in another when queries can carry along annotations from source data. We say that a query is *annotation-contained* in another if the annotated output of the former is contained in the latter on every possible annotated input databases. We study the relationship between query containment and annotation-containment and show that annotation-containment is a more refined notion in general. As a consequence, the usual equivalences used by a typical query optimizer may no longer hold when queries can carry along annotations from the source to the output. Despite this, we show that the same annotated result is obtained whether intermediate constructs of a query are evaluated with set or bag semantics. We also give a necessary and sufficient condition, via homomorphisms, that checks whether a query is annotation-contained in another. Even though our characterization suggests that annotation-containment is more complex than query containment, we show that the annotation-containment problem is NP-complete, thus putting it in the same complexity class as query containment. In addition, we show that the annotation placement problem, which was first shown to be NP-hard in [7], is in fact DP-hard and the exact complexity of this problem still remains open.

1 Introduction

Many databases that we can find on the Web today are derived from other (source) databases. New databases are often created because there is a need for "customized data" and often, these databases are curated with new data added in the process (some examples are [2, 3, 12]). In this situation, a system that is able to carry along superimposed information [23] or annotations about data as the data is being transformed is useful in many aspects as we shall describe shortly. We envision an annotation management system that is not only capable of carrying *forward* annotations of data as data is being transformed, but also capable of attaching new annotations on derived data *back* to the source data. Such bidirectional movement of annotations is useful for spreading information about a piece data consistently in a network of inter-dependent databases. Having such a systematic tool for sharing additional information about data automatically and dynamically via annotations is especially valuable in a collaborative environment where different views of the same database often exist simultaneously. The

[*] This research is supported by faculty research funds granted by the University of California, Santa Cruz.

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 37–53, 2004.

method by which an annotation is propagated forth and back in our system is based on *provenance* [6,7], more specifically *where-provenance*. Intuitively, the set of annotations associated with a piece of data d in the output is obtained from the annotations that are associated with every source data where d is *copied from*. The main feature of such provenance-based annotation management system is that if every piece of source data is annotated with its location (or identifier), then one can trace the provenance (or flow) of a piece of data d in the result by simply looking at its annotations (i.e. the set of all source locations where d is copied from) of d. By virtue of the ability of annotations to add value to data, annotations can be used creatively for a variety of other purposes. We describe some possible applications of our provenance-based annotation management system below.

To Keep Information That Cannot Stored According to the Current Database Design. Another use of annotations is to describe information about data (or meta-data) that would otherwise have not been kept in a database. For example, the additional information that an integer "50" in a database was measured "in ponds"[1] while for some others, "in kilograms" can be described through annotations, especially if the database schema is too rigid to accommodate such comments.

To Highlight Wrong Data. Our annotation framework can also be viewed as a semi-automatic tool for data cleaning. If an erroneous data was discovered in a database, an error report can be attached to that wrong data and propagated back to the source data. The maintainers of that source database can be notified of the error and therefore make the necessary corrections before the next database release. In the meanwhile, the error report can also be propagated to the same data in other databases, thus notifying other users of the error. Of course, one should also be able to add annotations to annotations. For example, the spelling mistake of the annotation "in ponds", described earlier, can be highlighted in another annotation.

Security. The security level of a data element can be described through annotations. When a query is executed, the annotations on the result of the query can be aggregated to determine the degree of sensitivity of the resulting output. This idea of using annotations to describe the security level of various data items or to specify fine-grained access control policies is not new and can be found in various forms in existing literature. E.g. [24,17,13].

Quality Metric. Annotations can also be used as a guide on the quality of data. The quality of a piece of data, based on some suitable metric, can be attached as an annotation. The aggregate of the annotations in the derived database can serve as an estimate to the overall quality of the derived database. We note that knowing the provenance of data can also serve as a guide to the overall quality of the derived database.

Other than using provenance as a basis for annotation propagations, there are conceivably other choices for the semantics of annotation propagation. As opposed to having such automatic mechanism for propagating and combining annotations, one could also imagine having explicit language constructs for dealing with annotations. We believe, however, that there are many applications, such as those described above, where

[1] "Pond" is intentionally misspelled.

such an automatic provenance-based annotation propagation is desirable. As we shall also describe in Section 5, this semantics of propagating annotations based on provenance is natural and can also found in existing literature [33, 21]. The containment properties of queries that can carry along annotations based on provenance, however, was never studied before.

Problem Statement and Summary of Main Results. In our framework, a binding associates a variable to a value as well as the set of annotations of that value, if any. With this definition of a binding, the method by which annotations are propagated from a source to the output follows the usual mechanics of query evaluation (by reasoning about its valuations), with the additional rule that the sets of annotations that are associated with the same output data are unioned together. Intuitively, two queries are *annotation-equivalent* if they produce the same annotated output on every annotated input databases. A query is *annotation-contained* in another if the result of the former query (an annotated database) is always contained in the result of the latter query under every possible annotated input database. An database D_1 is annotation-contained in another database D_2 if (1) every tuple in D_1 also occurs in D_2, and (2) the set of annotations associated with every column of every tuple in D_1 is a subset of the set of annotations that is associated with the same column of the same tuple in D_2.

We study the containment relationship among queries in our framework. We show that two equivalent queries are not annotation-equivalent in general. This result shows that traditional query optimization techniques cannot be immediately applied to our framework. In fact, given two equivalent conjunctive queries Q and Q', it is only when Q is a minimal query that one can determine that Q is annotation-contained in Q'. (A minimal query is a query in which no subquery, one that has less subgoals or joins, is equivalent to it.) Moreover, we show that there are other equivalences used by query optimizers (such as query rewriting using views and query minimization) that do not apply in the case of queries which can carry along annotations. We also give a necessary and sufficient condition, via homomorphisms, that characterizes whether a query is annotation-contained in another. Stating intuitively, the characterization shows that the problem of checking whether a query Q is annotation-contained in another query Q' is equivalent to the problem of finding a family of homomorphisms from Q' to Q. This family of homomorphisms ensures that Q' can simulate every possible way which Q may propagate annotations from the source to the output. Each homomorphism h in the family corresponds to an occurrence of a distinguished variable in the body of Q and its pre-image under h must be a distinguished variable in Q' that occurs "in the same way".

Such a homomorphism characterization is obviously more complex than its counterpart in classical query containment since it requires a family of homomorphisms to exist and each homomorphism is required to map each occurrence of a distinguished variable in a certain way. We show, however, that annotation-containment still belongs to the same complexity class as query containment, i.e., it is NP-complete. Moreover, despite having shown that usual query equivalences which are used by a query optimizer may no longer hold when queries can carry along annotations, we show that a query produces the same annotated result whether its intermediate constructs are evaluated with set or bag semantics. We also show that the *annotation placement problem*, which was first shown to be NP-hard in [7], is in fact DP-hard and conjecture that the exact complexity of this problem lies in a class that is slightly above DP. Given a query, a source database, the output, and a column of a tuple where the annotation is to be placed, the annotation placement problem is to determine whether there is an annotation that can be placed on some source data so that it only propagates to the the specified output data and no other places.

2 Preliminaries

Data Element, Location. A *data element* is the value of an attribute of a tuple in a relation. A *location* is a triple (R, t, i) consisting of a position number i (or an attribute), a tuple t, and a relation name R. For example, given the relation schema Person(ssn, name, age) and the instance { (123, "John", 34), (112, "Joe", 23), ... }, "John" is a data element of the tuple (123, "John", 34) under the attribute name and resides in the location (Person, (123, "John", 34), 2). We sometimes write a location as a pair in short, e.g., (Person(123, "John", 34), 2). An *annotation* is data attached to a data element, which resides in some location. There can be multiple annotations attached to the same data element. We sometimes use the phrase "an annotation is attached to a location" to mean the same as an annotation is attached to the data element that resides in that location. If there is an annotation "$*$" attached to the location (Person(112, "Joe", 23), 3), we will write the tuple as Person(112, "Joe", $23^{\{*\}}$).

Conjunctive Query. Our results are based on conjunctive queries [1]. A *conjunctive query* has the form $Q(\overline{X})$:-$S_1(\overline{Y_1}), ..., S_n(\overline{Y_n})$ where $\overline{X}, \overline{Y_i}, i \in [1, n]$ denote vectors of variables and every variable in $\overline{X}$ occurs in Y_i for some $i \in [1, n]$. Each $S_i(\overline{Y_i})$ is called a *subgoal* and each S_i where $i \in [1, n]$ is a relation name. The term $Q(\overline{X})$ is the *head* of the query. Variables in $\overline{X}$ are called *distinguished variables*. A conjunctive query is said to contain *views* if one or more of its subgoals is defined by the head of another conjunctive query. That is, for some $i \in [1, n]$, S_i is not a relation name and $S_i(\overline{Y_i})$ is the head of another conjunctive query. We use the notation $\overline{X}[i]$ to denote the ith variable in $\overline{X}$.

Semantics of Queries with Annotations. Intuitively, annotations are propagated from an input database to the result of a query according to the bindings of variables. For example, matching Person(X, Y, Z) against Person(112, "Joe", $23^{\{*\}}$) produces the valuation $\{X \rightarrow 112, Y \rightarrow$ "Joe"$, Z \rightarrow 23^{\{*\}}\}$. The set of annotations associated with the location (Person(112, "Joe", 23), age) is also bound to the variable Z. Therefore, if Z occurs in the head of the query, the annotation "$*$" appears at the location that corresponds to Z in the output under this valuation. Multiple annotations for the same output location are unioned together. For example, given the annotated database Person = { (318, "Jane", $23^{\{\circ\}}$), (112, "Joe", $23^{\{*\}}$)}, the query $Q(Z)$:- Person(X, Y, Z) produces the output $Q(23^{\{\circ,*\}})$. The precise semantics of conjunctive queries that propagate annotations is given in the appendix.

Correspondence of Locations, Annotation-Containment, and Equivalence. Let Q be a conjunctive query of the form $H(\overline{X}) : -S_1(\overline{Y_1}), ..., S_n(\overline{Y_n})$ where a subgoal of Q may be a view. Let D be a database and D' be the database D augmented with the (materialized) view predicates of Q. We say that a source location (s, i) in D *corresponds to* an output location (t, j) in $Q(D)$ according to Q and D if one of the following holds:

1. for some $k \in [1, n]$, $\overline{Y_k}[i] = \overline{X}[j]$, and there exists a valuation φ from Q into D' such that $H(\varphi(\overline{X})) = t$, $S_k(\varphi(\overline{Y_k})) = s$ (note that S_k is a relation name)
2. for some $k \in [1, n]$, S_k is a view, $\overline{Y_k}[p] = \overline{X}[i]$ for some position p, and there exists a valuation φ from Q into D' such that $H(\varphi(\overline{X})) = t$, and (s, i) corresponds to $(S_k(\varphi(\overline{Y_k})), p)$ according to V and D where V is the conjunctive query that defines the view $S_k(\overline{Y_k})$.

Continuing with the above example, (Person(318, "Jane", 23), 3) corresponds to (Q(23), 1). Similarly, (Person(112, "Joe", 23), 3) corresponds to (Q(23), 1). Notice that a database D_2 may be the result of a query Q applied on another database D_1. Annotation propagation is transitive. If a location l_1 in D_1 corresponds to l_2 in $Q(D_1)$ (i.e., D_2) and l_2 corresponds to l_3 in $Q'(D_2)$, then l_1 corresponds to l_3 according to $Q' \circ Q$ and D_1. Given a query Q and source database D, let $\mathcal{L}_{Q,D}$ denote the set of all pairs of locations (l, l') where l corresponds to l' under Q and D. Given two queries Q and Q', we say Q is *annotation-equivalent* to Q' if $\mathcal{L}_{Q,D} = \mathcal{L}_{Q',D}$ for every annotated input database D. We write $Q =_a Q'$ if Q and Q' are annotation-equivalent. Similarly, we write $Q \subseteq_a Q'$ if Q is *annotation-contained* in Q', i.e., $\mathcal{L}_{Q,D} \subseteq \mathcal{L}_{Q',D}$ for every annotated input database D. Notice that Q is trivially annotation-contained in Q' for any query Q' if Q is a boolean conjunctive query. For the rest of our discussion, we shall assume that our conjunctive queries are not boolean as it is this class of queries that is relevant to us, i.e., such queries can carry along annotations from the source to the result.

The proposition below shows that any conjunctive query with views can in fact be rewritten as a conjunctive query with only extensional database predicates (no views) in the body that is annotation-equivalent to the original query. We write Q^{exp} to denote the expansion of Q where all views of Q are replaced by their definitions so that every subgoal is an extensional database predicate. We shall assume that existentially quantified variables are distinct in the definitions of all the intensional database predicates.

Proposition 21 *If Q is a conjunctive query with views, then $Q^{exp} =_a Q$.*

3 The Behavior of Annotations under Query Evaluation

A query optimizer typically explores different but equivalent formulations of a given query before the best one is picked and executed. A natural question that arises is whether the same technique of picking the best equivalent query extends to queries which may carry along annotations, i.e., will the best plan produce the same annotated result?

We have found that it is possible for annotations to propagate differently even under simple equivalent conjunctive queries. This is not surprising if one observes that annotations can provide information about the "witnesses" (or source data) that is used to generate the output. Hence, annotations can be used to distinguish among equivalent queries since a piece of data in the result may have been generated for different reasons (in different ways) depending on the query, despite the fact that equivalent queries always arrive at same results under the same input database. In this section, we show that given two equivalent conjunctive queries Q and Q', neither $Q \subseteq_a Q'$ nor $Q' \subseteq_a Q$ in general. It is only under rather restricted circumstances that the equivalence of two queries imply annotation-containment. The converse is always true: whenever two queries are annotation-equivalent, they must be equivalent queries. Since the intermediate results of a query are often executed in bag semantics with duplicates removed only at the very end if needed, a related question that arises is whether query evaluation done in bag or set semantics would affect annotation propagations. We show that we obtain the same annotated result whether intermediate results of a query are evaluated with bag or set semantics. We give a necessary and sufficient condition for checking annotation-containment of queries and show that annotation-containment is NP-complete, which has equal complexity as the problem of deciding query containment. We also show that there are some other equivalences used by query optimizers that cannot be immediately applied to our framework.

3.1 Containment and Annotation-Containment

We first show that equivalent queries are not annotation-equivalent in general due to the use of equalities which are available in many query languages such as SQL.

Example 1. The following queries find pairs of employees of the same age (with the schema Emp(name, addr, age)).

$$Q_1 : \text{Ans}(X, X', Z) \text{ :- } \text{Emp}(X, Y, Z), \text{Emp}(X', Y', Z).$$
$$Q_2 : \text{Ans}(X, X', Z) \text{ :- } \text{Emp}(X, Y, Z), \text{Emp}(X', Y', Z'), Z{=}Z'.$$

Suppose we have the following tuples in the relation: Emp(John, Pine St, $30^{\{\star\}}$) and Emp(Joe, Walnut Ave, 30). The outputs of Q_1 and Q_2 are shown below (on the left and right respectively).

Ans(John, John, $30^{\{\star\}}$)	Ans(John, John, $30^{\{\star\}}$)
Ans(John, Joe, $30^{\{\star\}}$)	Ans(John, Joe, $30^{\{\star\}}$)
Ans(Joe, John, $30^{\{\star\}}$)	Ans(Joe, John, 30)
Ans(Joe, Joe, 30)	Ans(Joe, Joe, 30)

The difference in the results of Q_1 and Q_2 is due to the implicit equality, through variable Z, in Q_1 and the explicit equality $Z = Z'$ in Q_2. It appears that the above problem arises because annotations do not propagate across the equality operator. If annotations were exchanged between Z and Z' in Q_2 for every possible binding of values and annotations to Z and Z', then $Q_1 =_a Q_2$ in this example. We shall show, however, that even without the explicit use of equality, the behavior of annotations can still be different across equivalent queries. In other words, even if we had chosen the alternative semantics that annotations propagate across the equality operator, we would still run into the same problem. The next example shows that even without equalities, two equivalent queries are not annotation-equivalent in general. In fact, it is possible that neither of the queries is annotation-contained in the other.

Example 2. It can be easily verified that the queries below are equivalent.

Q_1: $\mathsf{Ans}(X, V)$:- $\mathsf{R}(X, Y, U)$, $\mathsf{R}(X, Z, V)$, $\mathsf{R}(T, W, Z)$.
Q_2: $\mathsf{Ans}(X, V)$:- $\mathsf{R}(P, Q, V)$, $\mathsf{R}(X, Z, V)$, $\mathsf{R}(T, W, Z)$.

Suppose we have the following annotated instance of R: $\mathsf{R}(1^{\{+\}}, 2, 3)$, $\mathsf{R}(1^{\{o\}}, 4, 5^{\{\star\}})$, $\mathsf{R}(1, 8, 4)$, and $\mathsf{R}(8, 9, 5^{\{\#\}})$. For Q_1, the result is $\mathsf{Ans}(1^{\{+,o\}}, 5^{\{\star\}})$. For Q_2, the result is $\mathsf{Ans}(1^{\{o\}}, 5^{\{\star,\#\}})$. Clearly, neither $Q_1 \subseteq_a Q_2$ nor $Q_2 \subseteq_a Q_1$. The difference in results shows that while two equivalent queries always arrive at the same answers given the same input database, the method by which they arrive at the answers can be rather different in general. Since annotation propagations are largely determined by the valuations that occur during evaluation, its propagation behavior is highly sensitive to the way the queries are written. This example also suggests that it is difficult to determine the annotation-containment relationship between two queries in general, even if they are equivalent. We can show, however, that a sufficient condition for $Q \subseteq_a Q'$ is when Q and Q' are equivalent queries and Q is minimal.

Theorem 1. *If Q and Q' are equivalent conjunctive queries and Q is minimal, then Q is annotation-contained in Q'.*

Proof. In other words, if $Q = Q'$ and Q is minimal, then $Q \subseteq_a Q'$. The proof uses the following result from [16]: If $\mathbf{C}$ is the core of a finite structure $\mathbf{A}$, then there is a homomorphism $h : \mathbf{A} \rightarrow \mathbf{C}$ such that $h(v) = v$ for every member v of the universe of $\mathbf{C}$. A structure $\mathbf{A}$ over the schema $\langle R_1, ..., R_k \rangle$ is a sequence $\langle A, R_1^{\mathbf{A}}, ..., R_k^{\mathbf{A}} \rangle$ where A is a non-empty set and $R_i^{\mathbf{A}}$ is the relation that interprets the relation symbol R_i. A structure $\mathbf{A}$ is a *substructure* of $\mathbf{B}$ = $\langle B, R_1^{\mathbf{B}}, ..., R_k^{\mathbf{B}} \rangle$ if $A \subseteq B$ and $R_i^{\mathbf{A}} \subseteq R_i^{\mathbf{B}}$ for all $i \in [1, k]$. The structure $\mathbf{A}$ is a proper substructure of $\mathbf{B}$ if $\mathbf{A}$ is a substructure of $\mathbf{B}$ and $R_i^{\mathbf{A}} \subset R_i^{\mathbf{B}}$ for some $i \in [1, k]$. A substructure $\mathbf{C}$ is the *core* of a structure $\mathbf{A}$ if there is a homomorphism from $\mathbf{A}$ to $\mathbf{C}$ and no homomorphism from $\mathbf{A}$ to a proper substructure of $\mathbf{C}$.

It follows from this result of [16] that there is a homomorphism h from Q' to the minimal query (or core) Q'_c of Q' such that h maps every variable in a subgoal of Q'_c to itself (by viewing the canonical instances of the queries as structures). Since Q and Q' are equivalent and Q is minimal, it follows that Q'_c and Q are isomorphic up to variable renaming. (We shall assume, for convenience, that Q'_c and Q are identical.) Given this property, we show that if $(l_1, l_2) \in \mathcal{L}_{Q,D}$ for any database D, then $(l_1, l_2) \in \mathcal{L}_{Q',D}$.

Suppose $(l_1, l_2) \in \mathcal{L}_{Q,D}$ for some database D and let $l_1 = (s, i)$ and $l_2 = (t, j)$ where s and t are tuples in D and $Q(D)$ respectively, and i and j are position numbers. By the definition of location correspondences, there exists a valuation μ for Q such that (1) $H(\mu(\overline{X})) = t$ where $H(\overline{X})$ is the head of Q, and (2) $S_k(\mu(\overline{Y_k})) = s$ where S_k is the kth subgoal of Q, and (3) $\overline{X}[j] = \overline{Y}[i]$. Since Q is minimal, consider the isomorphism g from Q to Q' which maps every subgoal in Q into its corresponding "identical" subgoal in Q' and the head of Q to the head of Q'. (The mapping g is the identity mapping since we assumed that Q'_c and Q are identical.) Therefore, $\mu \circ g^{-1} \circ h$ is a valuation for Q'. In particular, $H(\mu \circ g^{-1} \circ h(\overline{X'})) = H((\mu \circ g^{-1}(\overline{X'})) = H(\mu(\overline{X})) = t$, where $H(\overline{X'})$ is the head of Q' and $\overline{X'} = \overline{X}$ (since g is the identity mapping). Note that $S_k(g(\overline{Y_k}))$ is also a subgoal in Q'_c. Hence $S_k(\mu \circ g^{-1} \circ h(\overline{Y_k})) = S_l(\mu \circ g^{-1}(\overline{Y_k})) = S_k(\mu(\overline{Y_k})) = s$. Clearly, $\overline{Y_k}[i] = \overline{X'}[j]$ and so, $(l_1, l_2) \in \mathcal{L}_{Q',D}$.

The proof can be generalized to unions of conjunctive queries. See [31].

If Q is a minimal query, then for every valuation that produces an annotation in an output tuple according to Q, there is a similar valuation for Q' that will produce the same annotation on that output tuple: first take the homomorphism from Q' into its minimal query, then the isomorphism from its minimal query to Q, and then the valuation taken by Q.

The result above is tight in the sense that we can no longer determine if $Q \subseteq_a Q'$ given only the precondition that $Q \subseteq Q'$. The following example shows that if $Q \subseteq Q'$, then $Q \not\subseteq_a Q'$ and $Q' \not\subseteq_a Q$ in general, even when both Q and Q' are minimal queries.

Example 3. Suppose Q_1 and Q_2 are the following queries. Q_1: $\mathsf{Ans}(X)$:- $\mathsf{R}(X, Y)$, $\mathsf{S}(X, Y)$ and Q_2: $\mathsf{Ans}(X)$:- $\mathsf{R}(X, Y)$. Note that $Q_1 \subseteq Q_2$ and both queries are minimal. If the database instance consists of the following tuples $\mathsf{R}(1^{\{\star\}}, 2)$, $\mathsf{R}(1^{\{+\}}, 3)$, and $\mathsf{S}(1^{\{o\}}, 2)$, the results of the queries are $\mathsf{Ans}(1^{\{\star,o\}})$ and $\mathsf{Ans}(1^{\{\star,+\}})$ respectively. Clearly, $Q_1 \not\subseteq_a Q_2$ and $Q_2 \not\subseteq_a Q_1$.

It is also straightforward to see that if Q and Q' are equivalent queries and Q is a minimal query, $Q' \not\subseteq_a Q$ in general. Let Q_3 be the query $\mathsf{Ans}(X)$:- $\mathsf{R}(X, Y)$, $\mathsf{S}(X, Y)$, $\mathsf{R}(X, Z)$. Obviously, Q_1 and Q_3 are equivalent and Q_1 is minimal. The result of Q_3 is the following tuple $\mathsf{Ans}(1^{\{\star,+,o\}})$.

3.2 Query Evaluation with Annotations under Bag vs. Set Semantics

We show next that for conjunctive queries with views, the same annotations are carried to the result whether intermediate results of the query are evaluated with bag or set semantics. This invariance property is important since duplicate removal is an expensive operation and most query engines evaluate intermediate results under bag semantics and remove duplicates only at the end, if needed.

Let $Q^b(D)$ and $Q(D)$ denote the bag and set result of Q applied on D respectively. With $Q^b(D)$ (resp. $Q(D)$), every view predicate of Q^b (resp. Q) is evaluated in bag semantics (resp. set semantics). Let *Unique* be the operator that given an annotated database, possibly with duplicate tuples, merges the annotations of duplicate tuples together and removes duplicate tuples. That is, given a bag of annotated tuples B, *Unique*(B) returns the set of tuples in B and for every tuple $t \in$ *Unique*(B) and every position p in t, $\mathcal{A}(t, p) = \quad_{t' \in B} \mathcal{A}(t', p)$, where $t' = t$. We use the notation $\mathcal{A}(t, p)$ to denote the set of annotations at position p of tuple t. The proof is shown in [31] and can be generalized to unions of conjunctive queries.

Theorem 2. *Given a conjunctive query Q, the result of evaluating Q in bag semantics and then applying the* Unique *operator is the same as the result of evaluating Q in set semantics. That is,* Unique $\circ\, Q^b =_a Q$.

Example 2 and Example 3 suggest that annotation-containment cannot be characterized through query containment alone. In the next section, we give a necessary and sufficient condition that characterizes annotation-containment via a family of homomorphisms. This characterization provides insight to the reason why query containment, which requires only one homomorphism to exist from one query to another, is insufficient to characterize annotation-containment.

3.3 A Homomorphism Theorem for Annotation-Containment

In the following, we use $Q[p]$ to denote the pth subgoal of query Q where $p > 0$ and $Q[0]$ to denote the head of Q. This is not to be confused with $\overline{X}[i]$ which denotes the ith variable among $\overline{X}$. We use the notation var($Q[p]$) to denote the vector of variables at $Q[p]$.

Theorem 3. *Given two conjunctive queries Q and Q', $Q \subseteq_a Q'$ if and only if for every variable X that occurs at both the ith position of var($Q[0]$) and the jth position of var($Q[p]$) for some p, there exists a homomorphism h from Q' to Q such that*

1. *h maps the body of Q' into the body of Q and the head of Q' to the head of Q, and*
2. *the variable that occurs at the jth position of var($Q'[q]$) is identical to the variable at the ith position of var($Q'[0]$), where $Q'[q]$ is a pre-image of $Q[p]$ under h. That is, for some subgoal q, var($Q'[q]$)$[j]$ = var($Q'[0]$)$[i]$ and $h(Q'[q])$ = $Q[p]$.*

Proof. (*If*) Suppose $(l_1, l_2) \in \mathcal{L}_{Q,D}$ for some database D. Let $l_1 = (s, j)$ and $l_2 = (t, i)$ where s and t are tuples in D and $Q(D)$ respectively and i and j are valid position numbers for s and t respectively. By the definition $(l_1, l_2) \in \mathcal{L}_{Q,D}$, there exists a valuation μ for Q such that $\mu(Q[p]) = s$ for some p, $\mu(Q[0]) = t$, and var$(Q[p])[j]$ = var$(Q[0])[i]$. Let h be the homomorphism from Q' to Q with the above described properties (1)-(2) on the distinguished variable at var$(Q[0])[i]$ and var$(Q[p])[j]$. That is, h maps the body of Q' into the body of Q, the head of Q' to the head of Q, h maps $Q'[q]$ for some q to $Q[p]$ and var$(Q'[q])[j]$ = var$(Q'[0])[i]$. Clearly, $\mu \circ h$ is a valuation for Q' that produces t: $\mu \circ h(Q'[q]) = \mu(Q[p]) = s$, $\mu \circ\ h(Q'[0]) = \mu(Q[0])$, and var$(Q'[q])[j]$ = var$(Q'[0])[i]$. Therefore, $(l_1, l_2) \in \mathcal{L}_{Q',D}$.
(*Only if*) Suppose $Q \subseteq_a Q'$, we show that the above described homomorphisms exist. Fix for some p, i, j so that $Q[p]$ and $Q[0]$ are such that var$(Q[0])[i]$ = var$(Q[p])[j]$ for some positions i and j. Let C be the canonical instance induced by Q. That is, for every subgoal $S(Y_1, ..., Y_m)$ in Q, C contains the tuple $S(Y_1^c, ..., Y_m^c)$ where Y_k^c is a constant corresponding to the variable Y_k and C contains only these tuples. With C, it is easy to construct a valuation μ for Q by mapping the each variable Y_i in Q to its corresponding constant Y_i^c. It is easy to see that μ^{-1} is well-defined in this case.

Suppose $Q[p]$ denote the subgoal $S(Y_1, ..., Y_m)$ of Q and $Q[0]$ has the form $H(X_1, ..., X_n)$. (Note that $Q'[0]$ is the same as $Q[0]$ modulo variable renaming.) Let $l_1 = (S(Y_1^c, ..., Y_m^c), j)$ and $l_2 = (H(X_1^c, ..., X_n^c), i)$ where $Y_j = X_i$. Obviously, $(l_1, l_2) \in \mathcal{L}_{Q,C}$ and since $Q \subseteq_a Q'$, $(l_1, l_2) \in \mathcal{L}_{Q',C}$. By definition of $(l_1, l_2) \in \mathcal{L}_{Q',C}$, there exists a valuation φ for Q' so that subgoals of Q' are mapped into tuples in C, $Q'[0]$ is mapped to $H(X_1^c, ..., X_n^c)$, $\varphi(Q'[q]) = S(Y_1^c, ..., Y_m^c)$ for some q, and var$(Q'[q])[j]$ = var$(Q'[0])[i]$. Clearly, $\mu^{-1} \circ \varphi$ is a homomorphism from Q' to Q that maps the body of Q' to the body of Q, the head of Q' to the head of Q. Furthermore, $\mu^{-1} \circ \varphi(Q'[q]) = \mu^{-1}(S(Y_1^c, ..., Y_m^c)) = S(Y_1, ..., Y_m) = Q[p]$ and var$(Q'[q])[j]$ = var$(Q'[0])[i]$.

The proof can be generalized to unions of conjunctive queries. See [31].

Condition (1) ensures that every fact produced by Q is also produced by Q' (from the homomorphism theorem [8]). Condition (2) ensures that the source-to-target correspondence relation for Q, i.e. $\mathcal{L}_{Q,D}$, can be simulated by Q': every annotation carried by Q to the output can also be carried by Q' to the output "in the same way". It is obvious from the theorem that if Q is annotation-contained in Q', then there exists at least one homomorphism that maps the body of Q' to Q and head of Q' to Q respectively (recall that these are not boolean conjunctive queries). Hence, $Q \subseteq Q'$. We note that if $Q \subseteq_a Q'$, then $Q \subseteq Q'$.

Observe that a homomorphism with properties (1) and (2) is required to exist for each distinguished variable in each subgoal of Q. One natural question is whether there exists a *single* homomorphism that satisfies both properties for every distinguished variable in each subgoal of Q. The next example shows that such a single homomorphism does not exist in general.

Example 4. Suppose we have the following queries Q_1: Ans(X) :- R(X, Y), R(X, Z) and Q_2: Ans(X) :- R(X, Y). By applying Theorem 3, we can verify that Q_1 is annotation-contained in Q_2. Any homomorphism from Q_2 to Q_1, however, maps the subgoal of Q_2 to at most one subgoal of Q_1. Hence there cannot exist a single homomorphism that would simultaneously simulate the annotation propagations of both subgoals in Q_1 through the variable X.

Complexity of Annotation-Containment. Annotation-containment is characterized by Theorem 3, which states that one is required to establish a family of homomorphisms (with certain characteristics), one for each occurrence of a distinguished variable in the body of Q, in order to decide if one query is annotation-contained in another. We show that despite the added complexity given by the above characterization, annotation-containment has the same complexity as classical query containment.

Proposition 31 *It is* NP*-complete to decide if $Q \subseteq_a Q'$ given two conjunctive queries Q and Q'.*

Query Minimization. Query minimization is an important aspect of query optimization. The minimal equivalent form of a query is usually a better query in that it requires less number of joins and hence, less expensive to evaluate. It is known that given any conjunctive query, there is a unique minimal query [8], up to variable renaming. An approach to derive the minimal query from the original query is to eliminate redundant subgoals, one at a time. In case every subgoal cannot be removed, we have the minimal query. Otherwise, we continue to minimize the new query (with one subgoal removed).

Naturally, we would like to apply the same procedure to find a minimal annotation-equivalent query. Somewhat surprisingly, the above procedure of eliminating one subgoal at a time no longer works well when annotations are involved. It is possible that every subgoal of a query cannot be removed without losing annotation-equivalence to the original query and yet, when more than one subgoal is removed, the resulting query is annotation-equivalent to the original. The following example illustrates this scenario.

Example 5. It is easy to verify that Q is annotation-equivalent to Q_{min} and Q_{min} cannot be minimized further. By removing the first subgoal from Q, we obtain Q' which is not annotation-equivalent to Q (the same argument applies to every other subgoal removed from Q, and is not limited only to the first subgoal of Q). Note, however, that Q, Q', and Q_{min} are equivalent queries and Q_{min} is the minimal query.

Q: Ans(X) :- R(X, Z, V), R(X, U, Z), R(X, Z', T), R(X, S, Z').
Q': Ans(X) :- R(X, U, Z), R(X, Z', T), R(X, S, Z').
Q_{min}: Ans(X) :- R(X, Z', T), R(X, S, Z').

The above example suggests that to minimize a query and preserve annotation-equivalence, one needs to attempt to remove one or more subgoals at a time. We then check whether the resulting query is annotation-equivalent to the original and return the query where the maximum number of subgoals can be removed without losing annotation-equivalence. The question of whether there is a unique annotation-minimal query remains open.

Answering Queries Using Views. Some classical results in answering queries using views no longer applies as well when a query is capable of carrying along annotations. The result of Halevy et al [22] states that if a query Q has p subgoals and Q' is a minimal and complete rewriting of Q using a set of views $\mathcal{V}$ (meaning that Q' uses only view predicates of $\mathcal{V}$ in the body of Q' and is minimal), then Q' has at most p literals. We show here that there is no analogous result in our context. Given a conjunctive query Q with at most p subgoals and Q' is a minimal and complete rewriting of Q that is annotation-equivalent to Q using a set of views $\mathcal{V}$, it is no longer true that Q' contains at most p literals.

V1(X, Z, Z') :- R(X, Z, V), R(X, Z', T).
V2(X, Z) :- R(X, U, Z).
V3(X, Z') :- R(X, S, Z').

Consider Q_{min} of Example 5 where Q_{min} has 2 subgoals and suppose we have the above set of views $\mathcal{V}$. A complete, minimal, and annotation-equivalent rewriting of Q_{min} uses all three views (we denote the following rewriting as Q_r): Ans(X) :- V1(X, Z, Z'), V2(X, Z), V3(X, Z'). It is easy to verify that the expansion of Q_r is the query Q of Example 5 and therefore annotation-equivalent to Q_{min}. Note that Q_r has 3 subgoals and no proper subset of subgoals of Q_r, is annotation-equivalent to Q_{min}.

4 The Annotation Placement Problem

As described before, we would like an annotation management system that is also capable of propagating an annotation on a data element d in the result back to the source. Since d is often copied from data in many different places in the source, there is a question of where one should propagate an annotation on d back to the source. One method is, obviously, to propagate an annotation on d back to every piece of source data where d is copied from. Alternatively, we could propagate the annotation back to a piece of source data s such that no other output data elements are copied from s except d. In the latter case, the annotation on s is called a *side-effect-free* annotation because by placing an annotation on the source data s, that annotation would not appear in any other output data except d in the result. We believe that for a practical implementation of an annotation management system, the user should be given a choice of where to attach back an annotation and the system should suggest the side-effect-free annotation when there is one.

More formally, the *annotation placement setting* is a quadruple $(S,\ Q,\ Q(S),\ l)$, where S is a source database, Q is a monotone relational algebra query, $Q(S)$ the output, and l is a location in the view where the user wishes to place her annotation. Given an annotation placement setting, the *annotation placement problem* is to determine a source location l' to attach the annotation so that the annotation on l' corresponds to l and a minimum number of other output locations. In the event that l' corresponds only to the desired output location l and no other output location, we say an annotation on l' is a side-effect-free annotation.

The annotation placement problem for a class of relational queries was first shown to be NP-hard in [7]. We show that the annotation placement problem for the same class of queries is in fact DP-hard and conjecture that the exact complexity of this problem is slightly above DP. We also suggest alternative solutions that we may adopt for this problem and this class of queries.

We show that the problem of deciding whether there exists a side-effect-free annotation is DP-hard for a class of annotation placement settings where the query Q contains both project and join operators. A language is in the class DP [25] if it is equal to the intersection of a language in NP and a language in coNP (this class includes both NP and coNP). Many database problems lie in the class DP. For example, Cosmadakis [11] showed that the problem of determining if a database instance is the result of a query applied on an instance is DP-complete. Our proof of DP-hardness (shown in [31]) relies on a reduction from a known DP-complete problem called SAT-UNSAT [25]: given two Boolean formulas ϕ and ϕ', is it true that ϕ is satisfiable and ϕ' is unsatisfiable? (More specifically, our proof uses a variant of this problem that is also DP-complete, where both formulas are 3SAT formulas.)

Theorem 4. *Given an annotation placement setting $(S,\ Q,\ Q(S),\ l)$ where Q involves both project and join operators, the problem of deciding whether there is a side-effect-free annotation for l is* DP*-hard.*

A special case of a project and join query where a polynomial time algorithm exists is when every use of project retains the key of the input relations. Hence, it is possible to determine, through the key of the output tuple that contains the annotation, where an annotation should attach back in the source.

An alternative approach for queries that involve project and join operators would be to explicitly store the correspondences between source and output locations in the result. Assume that every source location contains an annotation that describes its address. We carry these address annotations along as $Q(S)$ is computed. Hence the set of address annotations in each output location tell which source locations correspond to it (i.e., its provenance). Suppose a user wishes to attach a remark (another annotation) to an output location l. In order to propagate that remark back to the source with minimum side-effects, we compute for each address annotation "$\star$" that occurs in l, the number of other locations in the output that also contains "$\star$". (This will be the number of side-effects if we propagate the remark back to the address indicated by "$\star$".) The address annotation in l that occurs at a minimum number of other output locations is where the remark should be attached in the source so that it would generate the least side-effects. This method works well assuming that the query engine exhaustively searches for every possible valuation of a query that may produce an output tuple and in doing so, it keeps the complete provenance of every output location. If, however, the query engine is "smart" enough to skip some valuations (knowing that it has already produced an output tuple), then it may miss some annotation propagation. The preceding observation suggests that if we chose an implementation of our annotation framework that modifies an existing query engine to carry along the address annotations, we must first ensure that the query engine always searches through every possible valuation.

5 Related Work

The idea of explicitly maintaining provenance by forwarding annotations along data transformations is not new and has been proposed in various forms in existing literature [33, 21, 4]. In fact, our annotation propagation rules which propagate annotations based on where-provenance are similar to those proposed in [33]. The difference from [33] is that we do not carry along the provenance of intermediate locations. There are several independent efforts in building annotation systems to support and manage annotations on text and HTML documents [20, 29, 32, 18, 27]. Recently, annotation systems for genomic sequences have also been built [5, 14, 19]. Research on annotations has been largely focused on system issues such as the scalability of design, distributed support for annotations, as well as designs that may avoid the use of specialized browser or Web servers. Laliberte and Braverman [20] discussed how to use the HTTP protocol to design a scalable annotation system for HTML pages. Schickler, Mazer, and Brooks [29] discussed the use of a specialized proxy module that would merge annotations from an annotation store onto a Web page that is being retrieved before sending it to the client browser. Such a design would not require a specialized browser or Web server to support and manage annotations. Annotea [32, 18] is a W3C effort to support annotations on any Web document. Annotations are stored on annotation servers based on an RDF annotation schema and uses XPointer for pinpointing locations on a Web document. The client, Amaya, is a specialized browser that can understand, communicate, and merge annotations residing in the annotation servers with Web documents. Phelps and Wilensky [26] also discussed the use of annotations with certain desirable properties on multivalent documents [28] which support documents of different media types, such as images, postscript, or HTML. DAS or Biodas [5, 14] and the Human Genome Browser [19] are specialized annotation systems for genomic sequence data. Like most other annotation system designs, there is one or more (distributed) annotation servers for storing annotations. These systems merge data from various sources to display it graphically to an end user.

The annotation systems described so far share a common model: there is a collection of base elements and every annotation refers to some part of a base element. Base elements are either retrieved in part or as a whole and they do not undergo complex transformations. The task of the annotation viewer is therefore simple: retrieve the relevant annotations (look for annotations in the annotation server(s) that refer to portions of the base element) and merge them with the base element it refers to. In this paper, we consider a system where annotations are made on relational data, proposed in [7]. Unlike Web pages, the rigid structure of relations makes it easy to describe the exact position where an annotation should be attached. In contrast to annotations on Web pages, however, our output is often the result of a complex transformation process. To the best of our knowledge, even though our annotation propagation framework is not entirely new, the semantics of queries and backward propagation of annotations in such a system have not been previously explored.

6 Conclusions and Open Issues

We have described an annotation propagation framework that has potential uses in many areas and this paper is only a preliminary investigation of some of the issues that arise in such a framework.

Many issues remain open. Our goal of understanding all these issues is to develop insights that will help us build a provenance-based annotation management system with a formal foundation. Although the problem of deciding whether two queries are annotation-equivalent is NP-complete, we conjecture that there exists polynomial time algorithms for the class of conjunctive queries with bounded treewidth [10]. The question of whether query minimization under annotation-equivalence is church-rosser remains open. For the annotation-placement problem, we conjecture that the exact complexity class is slightly above DP. We would also like to extend our annotation model and study to other models, such as the XML model, whereby tuples or relations can also be annotated. It would also be interesting to explore the precise relationship between containment of conjunctive queries under bag-semantics and annotation-containment. Both problems share some similarities: For example, some equivalences that hold under set-semantics no longer hold under bag-semantics (annotation-semantics). We also know that there are some differences: It is known that bag-containment of conjunctive queries is computationally harder than set-containment of conjunctive queries [9] while we have shown that the complexity of annotation-containment is the same as set-containment of conjunctive queries.

As mentioned briefly in Section 1, there are conceivably other choices for the semantics of annotation propagation other than propagating annotations based on provenance. As opposed to having such automatic mechanism for propagating and combining annotations, one could also imagine having explicit language constructs for dealing with annotations. We believe, however, that this choice of annotation propagation is a natural one and that there are many applications, such as tracing provenance, security, and quality control, where such an automatic provenance-based system for propagating annotations is desirable. A thorough investigation of alternative semantics or possibilities, however, is needed. We would also like to extend our investigation to the framework where annotations are propagated based on why-provenance.

It would also be interesting to investigate how we may allow annotations to be queried (in an implementation independent way) directly. For example, we may want to "return all persons whose date-of-birth has been incorrectly recorded" by finding tuples in the relation whose date-of-birth value has been annotated with an error report. One possibility is to extend the query language with constructs to query annotations. We observe that it is also possible to query annotations (without extending the query language) by first defining a view. For instance, if we assume that in an implementation, annotations are stored as another relation, one can define an XML view over the main database and annotation database so that annotations appear together with the data it annotates as part of the data in the XML view. Queries can then be posed over this view [30, 15], thus allowing annotations to be queried.

Acknowledgments

This paper would not have taken its current shape without many insightful comments and suggestions by Phokion G. Kolaitis. The author also thanks Lucian Popa and the reviewers for many helpful comments.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley Publishing Co, 1995.
2. A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research*, 28:45–48, 2000.
3. D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, B. Rapp, and D. Wheeler. GenBank. *Nucleic Acids Research*, 28(1):15–18, 2000.
4. P. Bernstein and T. Bergstraesser. Meta-Data Support for Data Transformations Using Microsoft Repository. *IEEE Data Engineering Bulletin*, 22(1):9–14, 1999.
5. biodas.org. http://biodas.org.
6. P. Buneman, S. Khanna, and W. Tan. Why and Where: A Characterization of Data Provenance. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 316–330, London, United Kingdom, 2001.
7. P. Buneman, S. Khanna, and W. Tan. On Propagation of Deletions and Annotations Through Views. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 150–158, Wisconsin, Madison, 2002.
8. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90, Boulder, Colorado, 1977.
9. S. Chaudhuri and M. Y. Vardi. Optimization of *real* conjunctive queries. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 59–70, Washington, DC, 1993.
10. C. Chekuri and A. Rajaraman. Conjunctive Query Containment Revisited. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 56–70, Delphi, Greece, 1997.
11. S. S. Cosmadakis. The Complexity of Evaluating Relational Queries. *Information and Control*, 58(1-3):101–112, 1983.
12. S. B. Davidson, J. Crabtree, B. P. Bunk, J. Schug, V. Tannen, G. C. Overton, and C. J. S. Jr. K2/Kleisli and GUS: Experiments in integrated access to genomic data sources - Author bios. *IBM Systems Journal*, 40(2):512–531, 2001.
13. D. E. Denning, T. F. Lunt, R. R. Schell, W. R. Shockley, and M. Heckman. The SeaView Security Model. In *IEEE Symposium on Security and Privacy*, pages 218–233, Washington, DC, 1988.
14. R. Dowell. A Distributed Annotation System. Technical report, Department of Computer Science, Washington University in St. Louis, 2001.
15. M. Fernández, Y. Kadiyska, A. Morishima, D. Suciu, and W. Tan. SilkRoute: A Framework for Publishing Relational Data in XML. *ACM Transactions on Database Systems (TODS)*, 27(4):438–493, 2002.
16. P. Hell and J. Nešetřil. The Core of a Graph. *Discrete Mathematics*, 109:117–126, 1992.
17. S. Jajodia and R. S. Sandhu. Polyinstantiation integrity in multilevel relations. In *IEEE Symposium on Security and Privacy*, pages 104–115, Oakland, California, 1990.

18. J. Kahan, M. Koivunen, E. Prud'Hommeaux, and R. Swick. Annotea: An open rdf infrastructure for shared web annotations. In *Proceedings of the International World Wide Web Conference(WWW10)*, pages 623–632, Hong Kong, China, 2001.
19. W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The Human Genome Browser at UCSC. *Genome Research*, 12(5):996–1006, 2002.
20. D. LaLiberte and A. Braverman. A Protocol for Scalable Group and Public Annotations. In *Proceedings of the International World Wide Web Conference(WWW3)*, Darmstadt, Germany, 1995.
21. T. Lee, S. Bressan, and S. Madnick. Source Attribution for Querying Against Semistructured Documents. In *Workshop on Web Information and Data Management (WIDM)*, Washington, DC, 1998.
22. A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 95–104, San Jose, California, 1995.
23. D. Maier and L. Delcambre. Superimposed Information for the Internet. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, pages 1–9, Philadelphia, Pennsylvania, 1999.
24. A. C. Myers and B. Liskov. A decentralized model for information control. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.
25. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
26. T. A. Phelps and R. Wilensky. Multivalent Annotaions. In *Proceedings of the First European Conference on Research and Advanced Technology for Digital Libraries*, pages 287–303, Pisa, Italy, 1997.
27. T. A. Phelps and R. Wilensky. Multivalent documents. *Proceedings of the Communications of the Association for Computing Machinery (CACM)*, 43(6):82–90, 2000.
28. T. A. Phelps and R. Wilensky. Robust intra-document locations. In *Proceedings of the International World Wide Web Conference(WWW9)*, pages 105–118, Amsterdam, Netherlands, 2000.
29. M. A. Schickler, M. S. Mazer, and C. Brooks. Pan-Browser Support for Annotations and Other Meta-Information on the World Wide Web. In *Proceedings of the International World Wide Web Conference(WWW5)*, Paris, France, 1996.
30. J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. *VLDB Journal*, 10(2-3):133–154, 2001.
31. W. Tan. Containment of Relational Queries with Annotation Propagation. Technical report, Dept. of Computer Science, University of California, Santa Cruz, 2003.
32. W3C. Annotea Project. http://www.w3.org/2001/Annotea.
33. Y. R. Wang and S. E. Madnick. A Polygen Model for Heterogeneous Database Systems: The Source Tagging Perspective. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 519–538, Brisbane, Queensland, Australia, 1990.

Appendix

Semantics of Query with Annotations. We let $[\![E]\!]^D_\mu$ denote the meaning of an expression E evaluated on database D under the context μ where μ is a valuation. Let $\mathcal{A}(l)$ denote the set of all annotations at location l and $\overline{X}[p]$ denote the pth variable in the vector of variables $\overline{X}$. Our goal is to define $[\![Q]\!]^D$ and $\mathcal{A}(l)$ where Q is a conjunctive query of the form $R(\overline{X})$:-$S_1(\overline{Y_1}), ..., S_n(\overline{Y_n})$ and l is a location in $Q(D)$. The variables in $\overline{X}$ occur in $\overline{Y_1}, ..., \overline{Y_n}$ and $n \geq 0$. If $n = 0$, then $\overline{X}$ must not contain any variables and $\llbracket Q \rrbracket^D = \{R(\overline{X})\}$. We shall assume that every subgoal is an extensional database predicate. If Q contains some variables $V_1, ..., V_m$, we first augment the database D with $V_1(D), ..., V_m(D)$ and compute Q against the augmented database. A valuation μ is *consistent* with another valuation ϕ if they agree on the assignments of common variables, i.e., if μ maps x to c_1 and ϕ maps x to c_2, then $c_1 = c_2$.

1. The denotation of a query.

   $\llbracket Q \rrbracket^D = \{R(\phi(\overline{X})) \mid \phi \in \llbracket S_1(\overline{Y_1}), ..., S_n(\overline{Y_n}) \rrbracket^D_{\{\}}, n \geq 1\}$

2. The set of annotations at an output location $(R(t), p)$.

   $\mathcal{A}(R(t), p) = \{a \mid \phi \in \llbracket S_1(\overline{Y_1}), ..., S_n(\overline{Y_n}) \rrbracket^D_{\{\}}, \phi(\overline{X}) = t,\ \overline{Y_i}[q] = \overline{X}[p],$
   $a \in \mathcal{A}(S_i(\phi(\overline{Y_i})), q))\}$

3. The denotation of a subgoal under context μ.

   $\llbracket S(\overline{Y}) \rrbracket^D_\mu = \{\phi \cup \mu \mid \phi$ is a valuation for variables in $\overline{Y},\ S(\phi(\overline{Y}))$ is true in D,
   and ϕ is consistent with $\mu\}$

4. The denotation of a sequence of subgoals under context μ.

   $\llbracket S_1(\overline{Y_1}), ..., S_k(\overline{Y_k})) \rrbracket^D_\mu = {}_{\phi} \llbracket S_2(\overline{Y_2}), ..., S_n(\overline{Y_n}) \rrbracket^D_{\mu \cup \phi}$ where ϕ maps variables in $\overline{Y_1}$ to values such that $S_1(\overline{Y_1})$ is true in D and ϕ is consistent with μ.

We note that Example 1 requires the above definition to be extended for equalities. This can be easily achieved by requiring the additional check, in definition (3) above, that $(\phi \cup \mu)(X) = (\phi \cup \mu)(X')$ for every equality $X = X'$ in Q.

Avoiding Unnecessary Ordering Operations in XPath

Jan Hidders and Philippe Michiels

University of Antwerp
Dept. of Mathematics and Computer Science
Middelheimlaan 1, BE-2020 Antwerpen, Belgium
{jan.hidders,philippe.michiels}@ua.ac.be

Abstract. We present a sound and complete rule set for determining whether sorting by document order and duplicate removal operations in the query plan of XPath expressions are unnecessary. Additionally we define a deterministic automaton that illustrates how these rules can be translated into an efficient algorithm. This work is an important first step in the understanding and tackling of XPath/XQuery optimization problems that are related to ordering and duplicate removal.

1 Introduction

The XQuery Formal Semantics [5] provide a full description of both XPath's [2] and XQuery's [3] semantics and an extensive set of rules for the translation of both languages into the XQuery Core language. The semantics of XPath [12] require that the result of an XPath expression (with exception of the sequence operator) is sorted by document order and duplicate-free. In addition, some XPath expressions — for instance, those that contain aggregate functions or element indices — also require that their input is duplicate-free and sorted. As a consequence many of the implementations that are faithful to the XPath semantics, such as Galax [6], insert an explicit operation for sorting and duplicate elimination after each step. These operations often create bottlenecks in the evaluation of certain XPath expressions on large documents. Therefore many other implementations omit these operations and sacrifice correctness for the sake of efficiency. In many cases however, these time consuming operations are not necessary because (1) after certain steps the result will always be sorted and/or duplicate-free or (2) the context in which this XPath expression occurs does not depend on the ordering or uniqueness of the nodes in the path expression's result.

The main contributions of this work are:

- A sound and complete set of inference rules that deduce whether an XPath expression that is evaluated by a straightforward implementation, that omits all sorting and duplicate elimination operations, always results in a list of unique nodes that is in document order.

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 54–70, 2004.

- The illustration of how these rules interact and how they can be used for the definition of an efficient algorithm realised by deterministic automata.

To understand why finding such a rule set is not trivial, consider the following two examples. The relative path expression `ancestor::*/foll-sibl::*/ parent::*` when evaluated for a single node, always produces an ordered result. However, its subexpression `ancestor::*/foll-sibl::*` clearly does not have that property. It is quite surprising to see that intermediate results are unordered whereas the final result is ordered.

One might think that the above only occurs after following certain axes. But this is not the case. Take, for instance, the path `child::*/parent::*/foll-sibl::*/parent::*`. Once again, this result of the expression always is sorted (which we will explain later). But the subexpression `child::*/parent::*/foll-sibl::*` sometimes produces a result out of document order.

The remainder of the paper is organized as follows. After defining some essential concepts in Section 2, Section 3 discusses a set of properties that we need for categorizing XPath expressions and for deducing the two essential properties: *order* and *duplicate-freeness*. These rules are defined in Section 4. In Section 5 we present deterministic automata that show the interactions between the rules and illustrate how our approach can be translated into an efficient algorithm. In Section 6, we discuss how our work can be applied to improve the performance of the Galax XQuery engine. Section 7 is about related work and we conclude in Section 8.

2 XPath

We begin with a (simplified) formalization of an XML document.

Definition 1 (XML Document). *An* XML document *is a tuple* $D = (N, \lhd, r, \lambda, \prec)$ *such that* $(N, \lhd)$ *is a directed graph that is a tree with root* r *and* $\lhd$ *giving the parent-child relationship,* $\lambda : N \to T$ *is a labeling of the nodes and* $\prec$ *is a strict total order over* N *that defines a preorder tree walk over a document tree. The relation* $\lhd^+$ *denotes the transitive closure of* $\lhd$*. The example document in Figure 1 shows the node identities inside the nodes. Their indices are numbered according to the document order and every node has its label near it.*

Note that we do not consider processing instructions, namespaces, text nodes or attributes here. Next, we define the syntax of the XPath fragment that we will consider.

Definition 2 (XPath Expression). *The language of* XPath expressions*, denoted as* $\mathcal{P}$*, is defined by the following abstract grammar*

$$
\begin{aligned}
P &::= A \mid P/A \\
A &::= \uparrow \mid \downarrow \mid \uparrow^{+} \mid \downarrow^{+} \mid \uparrow^{*} \mid \downarrow^{*} \mid \twoheadleftarrow \mid \twoheadrightarrow \mid \dot{\twoheadleftarrow} \mid \dot{\twoheadrightarrow}
\end{aligned}
$$

where A *is the set of all axes defined as follows*

symbol	**axis**
$\downarrow$	`child`
$\downarrow^{+}$	`descendant`
$\downarrow^{*}$	`descendant-or-self`
$\twoheadrightarrow$	`following`
$\dot{\twoheadrightarrow}$	`following-sibling`
$\uparrow$	`parent`
$\uparrow^{+}$	`ancestor`
$\uparrow^{*}$	`ancestor-or-self`
$\twoheadleftarrow$	`preceding`
$\dot{\twoheadleftarrow}$	`preceding-sibling`

This notation is an extension of the one used in [1] and is primarily used for compactness reasons. The `following` axis is defined as all nodes that are after the context node according to the document order, excluding any descendants and attributes. The `preceding` axis is defined analogously. The `following-sibling` nodes are those siblings of the context node that are after it according to the document order. The `preceding-sibling` axis selects those that are before the context node.

In XPath, step expressions are of the form `axis :: nodetest [predicate]`. Our syntax ignores predicates and node tests. For instance, the path expression $\downarrow/\uparrow$ actually represents the XPath expression `child::*/parent::*`.

Also, the `self` axis is disregarded here, because it represents the identity function and as a consequence preserves all properties of the preceding XPath expression. The grammar does not include the production rule P ::= P/P. This implies that we do not take into account path expressions of the form $p_1/(p_2/p_3)$. However, our theory can be generalized to include such expressions.

The semantics of an XPath expression p in a document D is denoted by the function

$$[\![p]\!]_D : N \to \mathcal{L}(N)$$

where N is the set of nodes in D and $\mathcal{L}(N)$ the set of all finite lists over N.

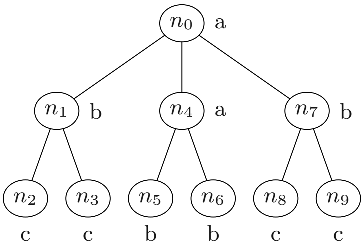


Fig. 1. In this example document, the index of the node id's inside the nodes correspond to the document order.

In addition to these formal semantics of path expressions, we define a "sloppy" semantics that corresponds to an implementation that does not eliminate duplicates and does not sort by document order after each step in the path expression. The semantics are defined by giving for each path expression p its "sloppy" implementation $\alpha(p)$ in terms of the XQuery core algebra [5].

The semantics of the implementation $\alpha(a)$ of an axis a is denoted by $[\![\alpha(a)]\!]_D$ and is equal to its formal semantics $[\![a]\!]_D$. Furthermore, we have for each path expression p and axis a

$$\alpha(p/a) = \text{for \$dot in } \alpha(p) \text{ return } \alpha(a)$$

In general, $[\![p]\!]_D \neq [\![\alpha(p)]\!]_D$. For instance, the quite simple path expression `//a/b/..` in the XPath semantics denotes the list $[n_0, n_4]$, if evaluated against the document in Figure 1. The sloppy semantics however, evaluates the very same expression into the list $[n_0, n_4, n_4, n_0]$, which is quite different.

The semantics of the sloppy implementation of a path expression p is called the *sloppy semantics* of p. It is easy to see that the sloppy semantics of p is equal to the formal semantics of p up to sorting and duplicate elimination. Whenever we talk about *the result* of an XPath expression, unless specified differently, we refer to the result under the sloppy semantics.

3 Path Properties

In this section, we introduce some properties of XPath expressions. These properties will assist us in determining whether the sloppy semantics of a path expression is equal to its formal semantics. In the next section, we define a set of rules for deriving these properties for each path expression.

The two main properties we want to derive for path expressions are

- *ord* - the *order* property, which indicates whether for all documents D and nodes n in D, $[\![\alpha(p)]\!]_D(n)$ is in document order (possibly with duplicates);
- *nodup* - the *no-duplicates* property, which indicates whether for all documents D and nodes n in D, $[\![\alpha(p)]\!]_D(n)$ contains no duplicates (but may not be in document order).

In order to derive these properties for all path expressions p we need an additional set of properties:

- *gen* - the *generation* property indicates whether for all documents D and nodes n in D all of the nodes in $[\![\alpha(p)]\!]_D(n)$ have the same distance to the root;

This property is a crucial factor for deciding whether the sloppy semantics of an XPath expression can have duplicate nodes. For instance, the path expressions $p/\downarrow^+$ and $p/\downarrow^*$, do not have the *nodup* property *unless* p has the *gen* property.

- $max1$ - the *max1* property indicates whether for all documents D and nodes n in D the length of the list $[\![\alpha(p)]\!]_D(n)$ is at most 1;

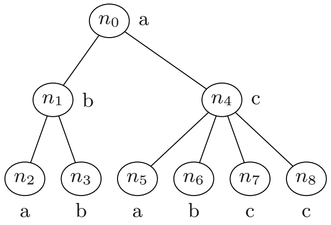


Fig. 2. Another simple XML document.

The path expression $\uparrow$ clearly has the $max1$ property and the path expression $\downarrow$ does not. For instance in Figure 2, $[\![\alpha(\downarrow)]\!]_D(n_1) = [n_2, n_3]$.

- *unrel* - the *unrelated* property indicates whether for all documents D and nodes n in D there are no two different nodes in $[\![\alpha(p)]\!]_D(n)$ that have an ancestor-descendant relationship; i.e., for all two nodes n_1 and n_2 in $[\![\alpha(p)]\!]_D(n)$ it holds that $n_1 \not\lhd^+ n_2$ and $n_2 \not\lhd^+ n_1$.

For instance, the path expression $\dot{\twoheadrightarrow}$ has the *unrel* property and $\uparrow^+$ does not. Note that *gen* implies *unrel* but not the other way around.

- ord_m - this property of a path expression p indicates whether for all documents D and nodes n in D, $[\![\alpha(p/\uparrow^m)]\!]_D(n)$ is in document order. This means that if we follow the $\uparrow$ axis m times from $[\![\alpha(p)]\!]_D(n)$, then the result will be in document order again. Obviously, ord_0 corresponds to *ord*.

The expression $\downarrow/\dot{\twoheadrightarrow}$ does not have the *ord* property, since for instance in Figure 2, $[\![\alpha(\downarrow/\dot{\twoheadrightarrow})]\!]_D(n_4) = [n_6, n_7, n_8, n_7, n_8, n_8]$, which is clearly not in document order. However, $\downarrow/\dot{\twoheadrightarrow}$ has the ord_1 property. Note that $[\![\alpha(\downarrow/\dot{\twoheadrightarrow}/\uparrow)]\!]_D(n_4) = [n_4, n_4, n_4, n_4, n_4, n_4]$.

- lin_m - this property of a path expression p indicates whether for all documents D and nodes n in D, $[\![\alpha(p/\uparrow^m)]\!]_D(n)$ is linear, i.e. for all two nodes $n_1, n_2 \in [\![\alpha(p/\uparrow^m)]\!]_D(n)$ it holds that $n_1 \lhd^+ n_2$, $n_2 \lhd^+ n_1$ or $n_1 = n_2$. We will use *lin* instead of lin_0.

The path expression $\uparrow^+$ has the property *lin*. Note that for each path expression p that has the *lin* and the *nodup* property, $p/\dot{\twoheadrightarrow}$ has the *unrel* property.

- sib_m - this property indicates whether for all documents D and nodes n in D, if a node k is in $[\![\alpha(p/\uparrow^m)]\!]_D(n)$ then all the left siblings of k or all the right siblings of k are in $[\![\alpha(p/\uparrow^m)]\!]_D(n)$. We will use *sib* instead of sib_0.

It is obvious that the expressions $\dot{\twoheadrightarrow}$ and $\dot{\twoheadleftarrow}$ have the *sib* property. This property plays a crucial role in the completeness proof of Section 5.

4 Inference Rules

We define a set of inference rules $\mathcal{R}$ for the deduction of the *nodup* and *ord* properties. The set of these rules is given in Figure 4. Not all rules are intuitive. We explain a few of them. If a path p has a certain property x, we will express this in the inference rules as $p : x$.

- The *gen* property is preserved by $\downarrow$, $\uparrow$, $\dot{\twoheadrightarrow}$ and $\dot{\twoheadleftarrow}$

  $$\frac{p : gen \qquad a \in \{\downarrow, \uparrow, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\}}{p/a : gen}$$

  This rule states that if a path expression p has the *gen* property i.e., all the nodes in the result of p have the same distance to the root, then if it is followed by one of the axes $\downarrow$, $\uparrow$, $\dot{\twoheadrightarrow}$, $\dot{\twoheadleftarrow}$, the entire expression has also the *gen* property.
- The *ord* property is preserved by the $\uparrow$ axis if the *gen* property also holds

  $$\frac{p : ord \qquad p : gen}{p/\uparrow : ord}$$

  If there are duplicates in $[\![\alpha(p/\uparrow)]\!]_D(n)$, then they are clustered since p has the *ord* property. Surprisingly, the *gen* property is absolutely required and cannot be replaced by the less restrictive *unrel* property.
  For instance, the path expression $\downarrow^+$ (evaluated from the node n_0 in the document in Figure 3) has the *ord* and *unrel* property, although $[\![\alpha(\downarrow^+/\uparrow)]\!]_D(n_0)$ is unordered.
  The above also implies that the removal of duplicates in this situation can be achieved very efficiently.

Theorem 1. *The rules in $\mathcal{R}$ are sound for the ord and nodup property; i.e., if we can derive with the rules in $\mathcal{R}$ in a finite number of steps that p : ord (p : nodup) for all XML documents D and nodes $n \in D$, it holds that $[\![\alpha(p)]\!]_D(n)$ is in document order (contains no duplicate nodes).*

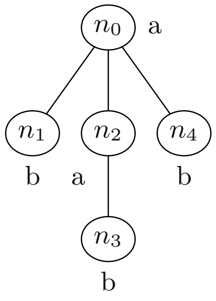


Fig. 3. An example XML fragment that shows the relevance of the *gen* property.

$$(1)\ \frac{p : \mathit{max}1}{p/\!\uparrow : \mathit{max}1} \qquad (2)\ \frac{p : \mathit{max}1}{p : \mathit{gen}} \qquad (3)\ \frac{\begin{array}{l} p : \mathit{gen} \\ a \in \{\downarrow, \uparrow, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \mathit{gen}}$$

$$(4)\ \frac{p : \mathit{gen}}{p : \mathit{unrel}} \qquad (5)\ \frac{p : \mathit{unrel}}{p/\!\downarrow : \mathit{unrel}} \qquad (6)\ \frac{p : \mathit{lin}_0 \qquad a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\}}{p/a : \mathit{unrel}}$$

$$(7)\ \frac{}{\downarrow : \mathit{unrel}} \qquad (8)\ \frac{\begin{array}{l} p : \mathit{max}1 \\ a \in \{\uparrow^+, \uparrow^*\} \end{array}}{p/a : \mathit{lin}_0} \qquad (9)\ \frac{p : \mathit{lin}_0}{p/\!\uparrow : \mathit{lin}_0}$$

$$(10)\ \frac{\begin{array}{l} p : \mathit{lin}_0 \\ a \in \{\downarrow, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \mathit{lin}_1} \qquad (11)\ \frac{p : \mathit{lin}_n \qquad n > 0}{p/\!\uparrow : \mathit{lin}_{n-1}} \qquad (12)\ \frac{\begin{array}{l} p : \mathit{lin}_n \qquad n > 0 \\ a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \mathit{lin}_n}$$

$$(13)\ \frac{p : \mathit{lin}_n}{p/\!\downarrow : \mathit{lin}_{n+1}} \qquad (14)\ \frac{p : \mathit{max}1}{p : \mathit{nodup}} \qquad (15)\ \frac{p : \mathit{max}1}{p/a : \mathit{nodup}}$$

$$(16)\ \frac{}{a : \mathit{nodup}} \qquad (17)\ \frac{p : \mathit{nodup}}{p/\!\downarrow : \mathit{nodup}} \qquad (18)\ \frac{\begin{array}{l} p : \mathit{nodup} \qquad p : \mathit{gen} \\ a \in \{\downarrow^+, \downarrow^*\} \end{array}}{p/a : \mathit{nodup}}$$

$$(19)\ \frac{\begin{array}{l} p : \mathit{lin}_0 \qquad p : \mathit{nodup} \\ a \in \{\uparrow, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \mathit{nodup}} \qquad (20)\ \frac{p : \mathit{max}1}{p/a : \mathit{ord}_0} \qquad (21)\ \frac{}{a : \mathit{ord}_0}$$

$$(22)\ \frac{\begin{array}{l} p : \mathit{ord}_0 \\ a \in \{\downarrow, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \mathit{ord}_1} \qquad (23)\ \frac{p : \mathit{ord}_0 \qquad n > 0}{p/\!\uparrow : \mathit{ord}_{n-1}} \qquad (24)\ \frac{\begin{array}{l} p : \mathit{ord}_n \qquad n > 0 \\ a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \mathit{ord}_n}$$

$$(25)\ \frac{p : \mathit{ord}_n}{p/\!\downarrow : \mathit{ord}_{n+1}} \qquad (26)\ \frac{\begin{array}{l} p : \mathit{unrel} \qquad p : \mathit{nodup} \\ p : \mathit{ord}_0 \\ a \in \{\downarrow, \downarrow^+, \downarrow^*\} \end{array}}{p/a : \mathit{ord}_0} \qquad (27)\ \frac{p : \mathit{ord}_0 \qquad p : \mathit{gen}}{p/\!\uparrow : \mathit{ord}_0}$$

$$(28)\ \frac{\begin{array}{l} a \in \{\downarrow, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}, \\ \twoheadrightarrow, \twoheadleftarrow, \downarrow^+, \downarrow^*\} \end{array}}{p/a : \mathit{sib}_0} \qquad (29)\ \frac{p : \mathit{sib}_n}{p/\!\downarrow : \mathit{sib}_{n+1}} \qquad (30)\ \frac{\begin{array}{l} p : \mathit{sib}_n \qquad n > 0 \\ a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p : \mathit{sib}_n}$$

$$(31)\ \frac{p : \mathit{sib}_n \qquad n > 0}{p/\!\uparrow : \mathit{sib}_{n-1}}$$

Fig. 4. The inference rules of $\mathcal{R}$ for determining the *nodup* and *ord* properties for expressions in $\mathcal{P}$.

Proof. (sketch[1]) To prove the theorem, we can prove soundness individually for each rule in $\mathcal{R}$.

5 Decision Procedure

The rules in $\mathcal{R}$ allow us to construct a deterministic automaton that decides whether or not the sloppy semantics of a path expression contains duplicates or is out of document order. To indicate that the algorithm can be easily implemented, we consider two separate automata: one for deriving the *nodup* property (A_{nodup}) and one for deriving the ord_0 property (A_{ord}). Both automata accept expressions p that have the *ord* (*nodup*) property, in a time that is linear to the length of p; i.e., the number of step expressions in p.

5.1 The A_{ord} Automaton

This infinite automaton (see Figure 5) shows five accept states. Each state is labeled with the properties that hold in that state. The three-dot symbols at the right indicate that the automaton has an infinite number of subsequent states with transitions from and to it that are the same as those of the last state before the symbol. The states are labeled with the same properties unless that property has an index. In this case, the index ascends in the subsequent states[2].

Note that the prefix of a path that has the *ord* property does not necessarily have the *ord* property itself; i.e., it is possible to return from an unordered state back into an ordered one.

Theorem 2. *A_{ord} is sound for the ord property; i.e., A_{ord} accepts only path expressions that have the ord property.*

Proof. (sketch) For each transition from state s_1 to state s_2, labeled with axis a, it holds that there is a set of inference rules in $\mathcal{R}$ that justifies it; i.e., if a path expression p has all properties associated with s_1, then p/a has all properties that are associated with s_2. Soundness then follows from the soundness of $\mathcal{R}$.

Theorem 3. *A_{ord} is complete for the ord property; i.e., every path expression that has the ord property is accepted by A_{ord}.*

Proof. (sketch) We first extend the automaton in shown Figure 5 as shown in Figure 8:

1. Add a single[3] *sink* state, which indicates that all extensions of expressions that lead to this state do not have the *ord* property;
2. For each state s in A_{ord} and for each axis a that does not have a transition from s, add a transition from s to the sink, labeled with a;

[1] For brevity, we will omit the proofs for the separate inference rules.

[2] This automaton can be readily implemented using a pushdown automaton or a finite automaton with a counter.

[3] In Figure 8, the sink state is displayed multiple times for readability.

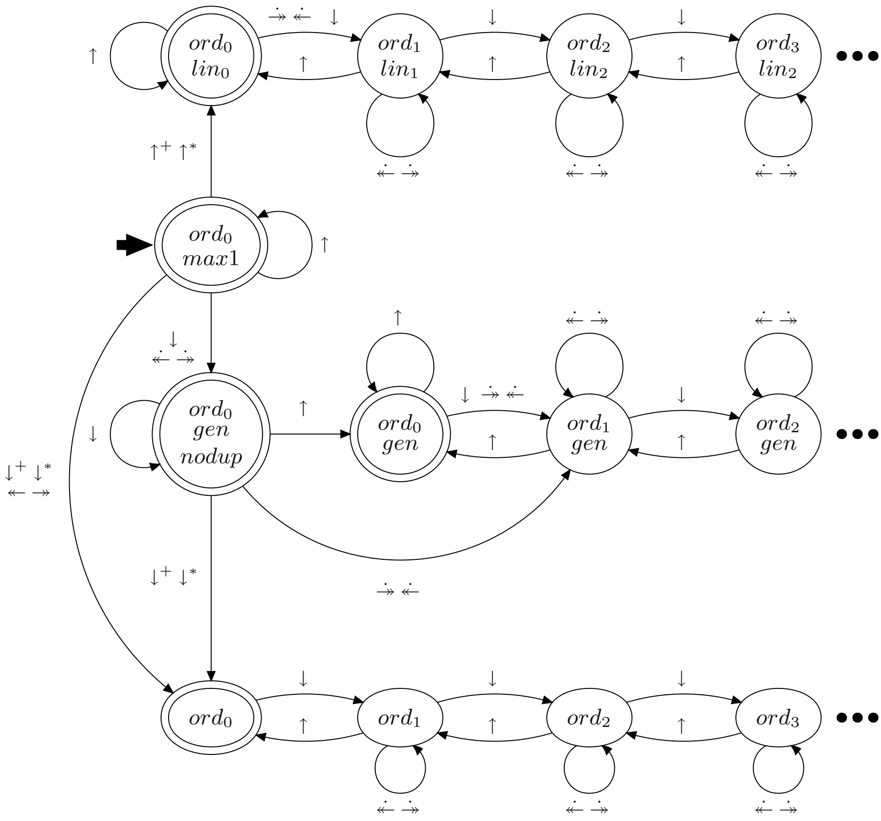


Fig. 5. The infinite A_{ord} automaton that decides the ord property.

3. We add labels to the old and new states of the automaton that indicate negative properties that hold in those states. Specifically, all the non-accepting states are labeled with the $\neg ord$ property, or one or more negative properties that imply this.

Note that $p : \neg ord$ does not mean that the result of p is always unordered. Instead it indicates that p does not have the ord property; i.e., the result of p may be unordered, depending on the document and context node for which p is evaluated. The automaton also has states with properties like

- $\neg ord_{\leq n}$, indicating that for all $i \leq n$ it holds that the path expression has the $\neg ord_i$ property;
- $\neg ord_{\geq n}$, indicating that for all $i \geq n$ it holds that the path expression has the $\neg ord_i$ property;

Next, we define an additional set of rules for the negative properties that justify the new transitions inside the new automaton. The automaton now defines for each state and for each axis a transition to another state. This means that

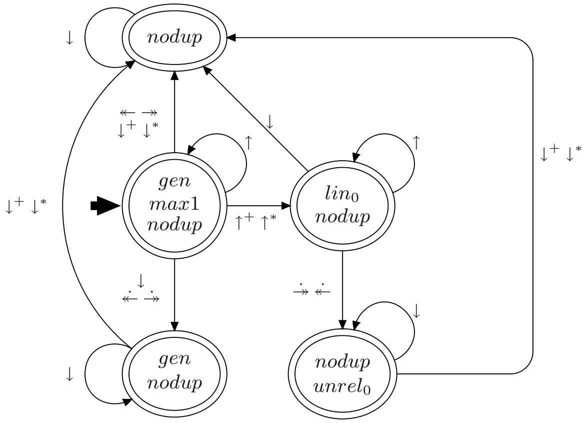

nodup
$\downarrow$
$\twoheadleftarrow$ $\twoheadrightarrow$
$\downarrow^+$ $\downarrow^*$
$\downarrow$
$\uparrow$
$\uparrow$
$\downarrow^+$ $\downarrow^*$
$\downarrow^+$ $\downarrow^*$
gen
max1
nodup
$\uparrow^+$ $\uparrow^*$
lin_0
nodup
$\downarrow$
$\dot{\twoheadleftarrow}$ $\dot{\twoheadrightarrow}$
$\dot{\twoheadrightarrow}$ $\dot{\twoheadleftarrow}$
$\downarrow$
$\downarrow$
gen
nodup
nodup
$unrel_0$

Fig. 6. The A_{nodup} automaton that decides the *nodup* property.

the automaton always ends in a certain state. By proving the additional rules to be sound, we know that all path expressions that lead to a non-accept state, indeed do not have the *ord* property and that all other expressions do. Thus every path expression that has the *ord* property is accepted by A_{ord}.

Figure 8 shows the result of extending our automaton. Note that in all accepting states, the ord_0 property holds and that in every other state, somehow the negation of this property holds. For instance $\neg ord_{\leq n}$ with $n \geq 0$ implies $\neg ord_0$. The new transitions and properties are justified by the rules given in Appendix A. If a path expression brings the automaton into a sink state, we know it's result will be unordered, no matter what the remainder of the expression is. For instance, if a path expression begins with $\downarrow^+/\uparrow$, then — no matter what step expressions follow — the entire expression will never regain the ord_0 property again.

5.2 The A_{nodup} Automaton

This finite automaton (see Figure 6) shows that, unlike the *ord* property, once the *nodup* property is lost, it never recurs; i.e., if a path expression has the *nodup* property, then so will any of its prefixes.

Theorem 4. *A_{nodup} is sound for the nodup property.*

Proof. Analogous to proof of Theorem 2.

Theorem 5. *A_{nodup} is complete for the nodup property.*

Proof. (sketch) Analogous to the proof of Theorem 3 (see Figure 9).

6 Implementation Strategy

We will use the Galax XQuery engine for evaluating our approach. Galax is one of the first implementations of XQuery and one of a few that implements XQuery's static type system. Galax's architecture is based on the XQuery Formal Semantics, making it an ideal platform for the implementation and evaluation of novel optimizations for XPath and XQuery.

In the previous sections, we have seen that ordering and duplicate elimination directly influence the evaluation efficiency of XPath. Indeed, unnecessary ordering and duplicate removal can cause a tremendous overhead during evaluation, especially when larger XML documents are involved. In the Galax [6] XQuery engine, for instance, this problem sometimes results in an unacceptable evaluation performance.

Our approach has an impact on the evaluation efficiency of most XPath expressions and, since XQuery is based on XPath, XQuery expressions can also profit from it. We can generalize our approach, working on the core expressions where we apply the optimizations on the `for`-loops into which XPath expressions are mapped.

Using our approach to determine whether a path expression generates duplicate nodes or nodes that are out of document order, we can optimize almost any XQuery Core mapping by eliminating any obsolete `distinct-docorder` operations. The `distinct-docorder` operation is a meta-operator that is inserted into the core expression during normalization in order to assure correctness regarding document order and no duplicates [5].

Take, for instance, the path expression

$$p = \texttt{/desc-or-self::b/c/foll-sibl::d/parent::*}$$

The formal semantics of p, $[\![p]\!]_D(n)$ is the list, without duplicates, of all nodes `c` that have a parent `b` and that have more than one `d` child. In Galax, this expression is mapped to the following, slightly simplified [4] core expression.

```
distinct-docorder(
  for $dot in $input
  return distinct-docorder(
    for $dot in desc-or-self::node()
    return distinct-docorder(
      for $dot in child::b
      return distinct-docorder(
        for $dot in child::c
        return distinct-docorder(
          for $dot in foll-sibl::d
          return parent::*
)))));
```

It is important to note that this core expression generates a duplicate-free list that is in document order.

However, the automata show that this expression is equivalent to the following one

```
distinct-nodes(
  for $dot in $input
  return
    for $dot in desc-or-self::b
    return
        for $dot in child::c
        return
          for $dot in foll-sibl::d
          return parent::*
)
```

Note that no sorting options are required in the query, where the original query had no less than six sorting operations. This actually means that the automata show that the sloppy semantics of p is equal to its formal semantics, up to duplicates. Additionally, since we know that the result of the query is in document order, we can remove duplicate nodes in linear time.

As the example shows, it may be useful to split the `distinct-docorder` operation into two separate instructions (one for sorting and one for eliminating duplicates from an ordered list) for cases where the result is ordered but contains duplicates.

There also seems to be an interesting interaction between this optimization technique and other schema-based optimizations. There are path expressions for which we cannot derive the *ord* or *nodup* properties. Nevertheless, when they are rewritten to equivalent expressions based on schema information [7, 10], it can sometimes become possible to derive these properties.

For example, if we consider the path expression `//b/c`, then the automata show that its sloppy semantics has nodes out of document order. But if we know from the schema of the source XML document that an element `b` only occurs nested inside an element `a`, which is the root then we can substitute `//b` with `/a/b`. Since this path contains only `child` axis, we can avoid an ordering operation. This technique can be used for optimizing path expressions that have any axes in them that do not preserve the *gen* property.

7 Related Work

Galax is not the only implementation facing these problems. In an attempt to pipeline step expressions from an XPath expression, [9] proposes a technique that avoids the generation of duplicate nodes in the first place. This is done by translating XPath expressions into a sequence of algebraic operations such that no duplicate nodes are generated, which is very important because the elimination of duplicates is a pipeline breaker. One of the basics of this approach is the rewriting of XPath expressions into step functions that do not generate duplicates. The preparatory rewriting rules used for this approach are inspired by [11] where the setup is to translate paths with reverse axis into equivalent ones, containing only forward axes. This approach is quite similar to the approach described in [8], where a pre and post numbering of the instance tree is used to avoid the generation of duplicates. However, in contrast with our algorithm, these approaches both have the slight disadvantage that the `position` and `last` variables cannot be used in the predicates of XPath expressions.

8 Conclusion and Future Work

Our approach has focussed primarily on two properties of XPath expressions: order and duplicate-freeness. We have shown for our XPath fragment, that we can efficiently derive whether a query evaluated by the *sloppy* implementation α, returns a result in document order and free from duplicates. This knowledge can be used to remove unnecessary ordering or duplicate removal operations from the query plan or to rewrite certain expressions so that neither ordering nor duplicate removal are required, like the schema based optimizations we discussed in section 6.

We will implement our algorithm into the Galax XQuery engine, where unnecessary `distinct-docorder` operations sometimes cause unacceptable evaluation performance for queries on large documents. The optimizer will be extended with an algorithm that manipulates the abstract syntax tree of XQuery core expressions to remove unnecessary ordering and duplicate removal operations. We expect that our approach will be very helpful in improving the performance of query evaluation however, not all unnecessary ordering or duplicate removal operations are removed. One reason for this is that we fail to take into account possible ordering or duplicate removal operations on a part of a query that influence the entire query. A simple example can illustrate this.

Consider for instance, the query

$$\downarrow / \uparrow / \downarrow / \downarrow / \uparrow$$

Suppose that an ordering operation is performed before the last step operator; i.e. after the evaluation of the subexpression $\downarrow / \uparrow / \downarrow / \downarrow$. If the result of this subexpression is being explicitly ordered, then the evaluation of the last $\uparrow$ axis will also yield an ordered result. This is because the ordering operation brings us back to the last state where the *ord* property did hold. Since our automaton has no notion of the ordering operation, this information gets lost and unnecessary ordering operations remain in the query execution plan.

In a next step we will extend our approach to detect these unnecessary operations and which will enable us to further optimize XPath evaluation.

Acknowledgements

We would like to thank the Galax team, especially Mary Fernández and Jérôme Siméon for introducing us to the problems discussed in this work, the many inspiring discussions and their suggestions for improvement.

References

1. M. Benedikt, W. Fan, and G. Kuper. Structural Properties of XPath Fragments. In *Proc. of ICDT'03, 2003.*, 2003.
2. Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0, W3C Working Draft 02 May 2003, 2003. `http://www.w3.org/TR/xpath20`.
3. Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0 An XML Query Language, W3C Working Draft 02 May 2003, 2003. `http://www.w3.org/TR/xquery`.
4. Byron Choi, Mary Fernández, and Jerôme Siméon. The XQuery Formal Semantics: A Foundation for Implementation and Optimization. Internal Working Document at AT&T and Lucent. `http://www.cis.upenn.edu/~kkchoi/galax.pdf`, 2002.
5. Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jerôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics, W3C Working Draft 2 May 2003, 2002. `http://www.w3.org/TR/query-semantics`.
6. M. Fernández and J. Siméon. *Galax, the XQuery implementation for discriminating hackers.* AT&T Bell Labs and Lucent Technologies, v0.3 edition, 2003. `http://www-db-out.bell-labs.com/galax`.
7. M. Fernández and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proceedings of the International Conference on Data Engineering*, pages 14–24, 1998.
8. Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *VLDB 2003*, 2003.
9. Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Optimized Translation of XPath into Algebraic Expressions Parameterized by Programs Containing Navigational Primitives. In *Proc. of the 3rd International Conference on Web Information Systems Engineering (WISE 2002)*, pages 215–224, Singapore, 2002.
10. April Kwong and Michael Gertz. Schema-based Optimization of XPath Expressions. Technical report, Univ. of California, dept. of Computer Science, 2001.
11. Dan Olteanu, Holger Meuss, Tim Furche, and Francois Bry. XPath: Looking Forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*, volume 2490 of *LNCS*, pages 109–127. Springer, 2002.
12. P. Wadler. Two semantics for XPath, 1999. `http://www.cs.bell-labs.com/who/wadler/topics/xml.html`.

A Rules for Negative Properties

Figure 7 shows the rules for the negative properties that are used for justifying the transitions in the extended automata (Figures 8 and 9). Proving the soundness for these rules is essential for proving completeness of the rules in $\mathcal{R}$.

B Extended Automata

Figure 8 shows the A_{ord} automaton, extended according to the algorithm described in the proof of Theorem 3 in Section 5. Figure 9 shows the same extension for the A_{nodup} automaton.

$$(32)\ \frac{a \in \{\downarrow^+, \downarrow^*, \twoheadrightarrow, \twoheadleftarrow\}}{p/a : \neg ord_{\geq 1}} \qquad (33)\ \frac{p : \neg ord_{\geq n}}{p/\downarrow : \neg ord_{\geq n+1}} \qquad (34)\ \frac{\begin{array}{l} p : \neg ord_{\geq n} \\ n > 0 \end{array}}{p/\uparrow : \neg ord_{\geq n-1}}$$

$$(35)\ \frac{\begin{array}{l} p : \neg ord_{\geq n} \\ n > 0 \\ a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \neg ord_{\geq n}} \qquad (36)\ \frac{\begin{array}{l} p : \neg max1 \\ a \in \{\uparrow^+, \uparrow^*, \twoheadrightarrow, \twoheadleftarrow\} \end{array}}{p/a : \neg ord_{\geq 0}} \qquad (37)\ \frac{p : \neg ord_{\geq 0}}{p/\uparrow : \neg ord_{\geq 0}}$$

$$(38)\ \frac{p : \neg ord_{\geq 0}}{p/\downarrow : \neg ord_{\geq 1}} \qquad (39)\ \frac{\begin{array}{l} p : \neg ord_{\geq 1} \\ p : \neg ord_{\leq n} \end{array}}{p : \neg ord_{\geq 0}} \qquad (40)\ \frac{\begin{array}{l} p : \neg ord_{\geq 0} \\ a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p : \neg ord_{\geq 0}}$$

$$(41)\ \frac{\begin{array}{l} p : \neg nodup \\ a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}, \downarrow, \downarrow^+, \downarrow^*\} \end{array}}{p/a : \neg ord_{\leq 0}} \qquad (42)\ \frac{\begin{array}{l} p : sib_0 \\ a \in \{\dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\} \end{array}}{p/a : \neg ord_{\leq 0}} \qquad (43)\ \frac{\begin{array}{l} p : \neg ord_{\leq 0} \\ a \in \{\downarrow, \dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}, \downarrow^+, \downarrow^*\} \end{array}}{p/a : \neg ord_{\leq 0}}$$

$$(44)\ \frac{p : \neg ord_{\geq 0}}{p : \neg ord_{\leq 0}} \qquad (45)\ \frac{\begin{array}{l} p : \neg unrel_0 \\ a \in \{\downarrow, \uparrow^+, \uparrow^*, \dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\} \end{array}}{p/a : \neg ord_{\leq 0}} \qquad (46)\ \frac{p : \neg ord_{\leq n}}{p/\downarrow : \neg ord_{\leq n+1}}$$

$$(47)\ \frac{\begin{array}{l} p : \neg ord_{\leq n} \\ n > 0 \end{array}}{p/\uparrow : \neg ord_{\leq n-1}} \qquad (48)\ \frac{\begin{array}{l} p : \neg ord_{\leq n} \\ n > 0 \\ a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \neg ord_{\leq n}} \qquad (49)\ \frac{p : \neg ord_{\leq n}}{p : \neg ord_{\leq 0}}$$

$$(50)\ \frac{\begin{array}{l} a \in \{\uparrow^+, \downarrow^+, \uparrow^*, \\ \downarrow^*, \twoheadrightarrow, \twoheadleftarrow\} \end{array}}{p/a : \neg gen} \qquad (51)\ \frac{\begin{array}{l} p : \neg gen \\ a \in \{\downarrow, \uparrow, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \neg gen} \qquad (52)\ \frac{\begin{array}{l} a \in \{\downarrow, \downarrow^+, \downarrow^*, \uparrow^+, \\ \uparrow^*, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}, \twoheadrightarrow, \twoheadleftarrow\} \end{array}}{p/a : \neg max1}$$

$$(53)\ \frac{p : \neg max1}{p/\uparrow : \neg max1} \qquad (54)\ \frac{p : \neg gen}{p/a : \neg max1} \qquad (55)\ \frac{\begin{array}{l} a \in \{\downarrow^+, \downarrow^*, \uparrow^+, \\ \uparrow^*, \twoheadrightarrow, \twoheadleftarrow\} \end{array}}{p/a : \neg unrel_0}$$

$$(56)\ \frac{\begin{array}{l} p : \neg unrel_0 \\ a \in \{\uparrow, \downarrow\} \end{array}}{p/a : \neg unrel_0} \qquad (57)\ \frac{\begin{array}{l} p : \neg unrel_0 \\ a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \neg unrel_1} \qquad (58)\ \frac{\begin{array}{l} p : \neg unrel_n \\ n > 0 \end{array}}{p/\uparrow : \neg unrel_{n-1}}$$

$$(59)\ \frac{p : \neg unrel_n}{p/\downarrow : \neg unrel_{n+1}} \qquad (60)\ \frac{\begin{array}{l} p : \neg unrel_n \\ n > 0 \\ a \in \{\dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \neg unrel_n} \qquad (61)\ \frac{\begin{array}{l} p : sib \\ a \in \{\uparrow, \dot{\twoheadrightarrow}, \dot{\twoheadleftarrow}\} \end{array}}{p/a : \neg nodup}$$

$$(62)\ \frac{p : \neg nodup}{p/a : \neg nodup} \qquad (63)\ \frac{\begin{array}{l} p : \neg max1 \\ a \in \{\uparrow^+, \uparrow^*, \twoheadleftarrow, \twoheadrightarrow\} \end{array}}{p/a : \neg nodup} \qquad (64)\ \frac{\begin{array}{l} p : \neg gen \\ p : \neg unrel_0 \\ a \in \{\downarrow^+, \downarrow^*\} \end{array}}{p/a : \neg nodup}$$

$$(65)\ \frac{}{p/\downarrow : \neg lin_0}$$

Fig. 7. The rules for the negative properties justify the transitions in the extended automata.

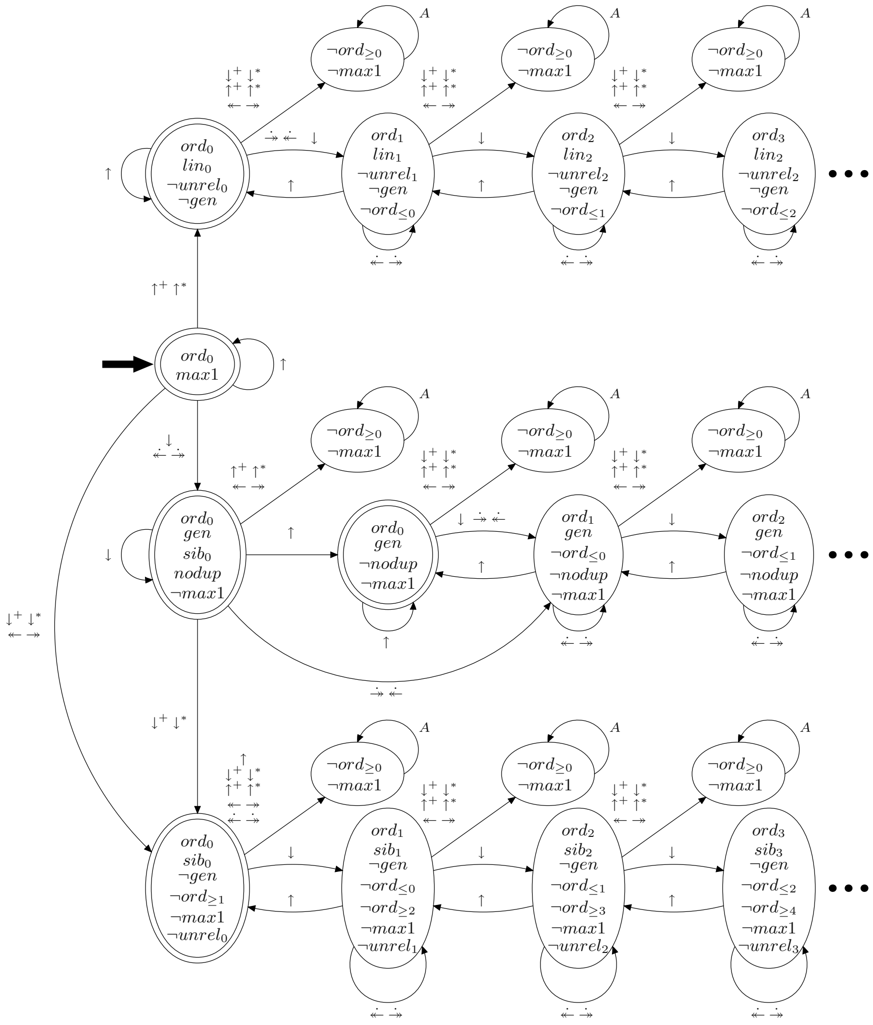

A
$\neg ord_{\geq 0}$ $\neg max1$
$\downarrow^+ \downarrow^*$ $\uparrow^+ \uparrow^*$ $\twoheadleftarrow \twoheadrightarrow$
ord_0 lin_0 $\neg unrel_0$ $\neg gen$
$\dot{\twoheadrightarrow} \dot{\twoheadleftarrow}$ $\downarrow$
$\uparrow$
ord_1 lin_1 $\neg unrel_1$ $\neg gen$ $\neg ord_{\leq 0}$
ord_2 lin_2 $\neg unrel_2$ $\neg gen$ $\neg ord_{\leq 1}$
ord_3 lin_2 $\neg unrel_2$ $\neg gen$ $\neg ord_{\leq 2}$
$\dot{\twoheadleftarrow} \dot{\twoheadrightarrow}$
$\uparrow^+ \uparrow^*$
ord_0 $max1$
$\downarrow$ $\dot{\twoheadleftarrow} \dot{\twoheadrightarrow}$
$\uparrow^+ \uparrow^*$ $\twoheadleftarrow \twoheadrightarrow$
ord_0 gen sib_0 $nodup$ $\neg max1$
ord_0 gen $\neg nodup$ $\neg max1$
$\downarrow$ $\dot{\twoheadrightarrow} \dot{\twoheadleftarrow}$
ord_1 gen $\neg ord_{\leq 0}$ $\neg nodup$ $\neg max1$
ord_2 gen $\neg ord_{\leq 1}$ $\neg nodup$ $\neg max1$
$\downarrow^+ \downarrow^*$ $\twoheadleftarrow \twoheadrightarrow$
$\downarrow^+ \downarrow^*$
$\uparrow$ $\downarrow^+ \downarrow^*$ $\uparrow^+ \uparrow^*$ $\twoheadleftarrow \twoheadrightarrow$ $\dot{\twoheadleftarrow} \dot{\twoheadrightarrow}$
ord_0 sib_0 $\neg gen$ $\neg ord_{\geq 1}$ $\neg max1$ $\neg unrel_0$
ord_1 sib_1 $\neg gen$ $\neg ord_{\leq 0}$ $\neg ord_{\geq 2}$ $\neg max1$ $\neg unrel_1$
ord_2 sib_2 $\neg gen$ $\neg ord_{\leq 1}$ $\neg ord_{\geq 3}$ $\neg max1$ $\neg unrel_2$
ord_3 sib_3 $\neg gen$ $\neg ord_{\leq 2}$ $\neg ord_{\geq 4}$ $\neg max1$ $\neg unrel_3$
$\bullet\bullet\bullet$

Fig. 8. The extended version of the A_{ord} automaton.

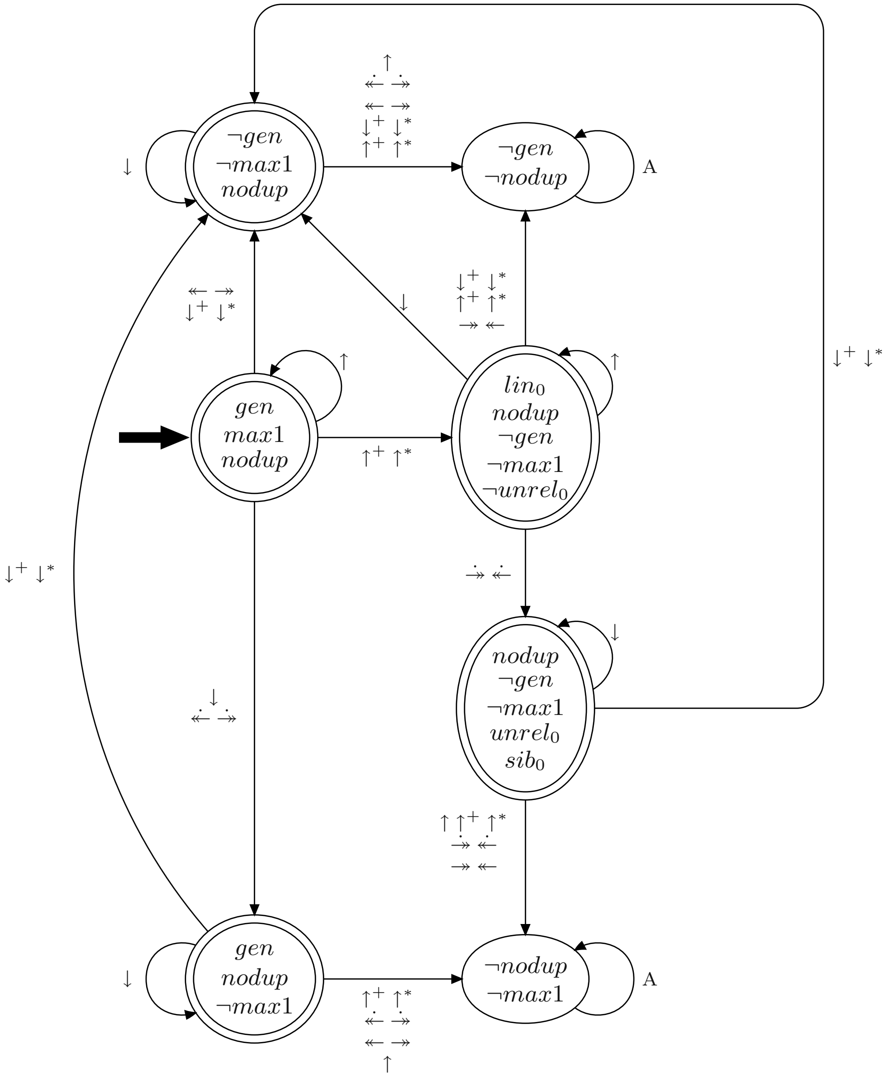


Fig. 9. The extended A_{nodup} automaton.

Consistency of Java Transactions*

Suad Alagić and Jeremy Logan

Department of Computer Science
University of Southern Maine
Portland, ME 04104-9300, USA
alagic@cs.usm.maine.edu

Abstract. A model of Java transactions is presented and the basic formal properties of this model with respect to behavioral compatibility and database integrity are proved. An implementation technique for incorporating constraints into the existing Java technology is developed, and a sophisticated theorem prover technology is integrated into this environment capable of enforcing behavioral compatibility and verifying transaction safety. The model of Java transactions developed in this paper is based on the subtle interplay of constraints, bounded parametric polymorphism and orthogonal persistence. The approach is based on a correct solution for extending Java with parametric polymorphism along with a technique for integrating logic-based constraints into Java class files and class objects. The Java Core Reflection is extended accordingly. The technique for enforcing database integrity constraints is based on the PVS theorem prover technology, and it combines static and dynamic checking to attain reasonable performance. The theorem prover technology is itself based on a sophisticated type system equipped with a form of semantic subtyping and parametric polymorphism. The higher-order features of this technology are proved to be critical in dealing with the problems of behavioral compatibility and transaction verification.

1 Introduction

This paper presents a model of Java transactions, establishes the basic formal properties of this model with respect to database integrity, and develops two specific results. One of them provides a solution for integrating logic-based constraints into the existing Java technology. The other one is the integration with a sophisticated theorem prover technology that is capable of enforcing behavioral compatibility and verifying transaction safety with respect to database integrity.

ACID properties refer to Atomicity (a transaction is executed as a unit or not at all), Consistency (a transaction is a unit of integrity, i.e., it maintains database integrity constraints), Isolation (a transaction is protected from actions of concurrent transactions with respect to database integrity) and Durability (once

* This material is based upon work supported in part by NSF under grant number IIS-9811452 and in part by the Institute for Information Sciences of the University of Southern Maine.

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 71–89, 2004.

committed the transaction's effects persist until changed by another transaction). No model of ACID Java transactions exists at present and certainly no implementation either. The most critical reason is that no existing model of Java transactions is capable of enforcing database integrity constraints that are specified in a declarative, possibly logic-based style, as customary in data models. This is where the main contributions of this paper are. Related results and projects as a rule provide atomicity and durability. All of them have nontrivial problems in providing isolation. But not a single related result or project is addressing properly the "C" component of ACID Java transactions, which is precisely the focus of this paper.

Of course, the fact that a model of ACID Java transactions has not been available as of yet lies in the limitations of Java itself. Java is completely lacking any kind of declarative, logic-based constraint capabilities. These constraints appear as assertions of object-oriented languages (such as Eiffel) and integrity constraints in database systems. Although this is the core of the problem, careful investigation reveals that other limitations that Java has as a database programming language also play an important role. This paper shows that a proper form of parametric polymorphism is essential for a sophisticated model of ACID Java transactions. A suitable model of persistence in Java (which the existing one is not) is naturally also required to have a correct model of database transactions. While a variety of results related to parametric polymorphism and persistence in Java is available, the situation is quite different when it comes to *persistent* Java constraints.

Although the focus of this paper is on problems that belong to database and persistent object-oriented systems, the paper makes an important contribution to the Java technology in general. This contribution is a technique for integrating declarative, logic-based constraints along with parametric classes into the Java environment. Such a technique is obviously required in order to address properly the problems of ACID Java transactions in particular. The technique offers correct run-time type information as reported by Java Core Reflection and thus correct persistent type information. Furthermore, this technique incorporates constraints into the Java environment. Both parametric classes and class constraints are represented in Java class files and class objects as they are defined and implemented in the existing Java Virtual Machine. Java Core Reflection is extended accordingly.

A viable technology for static verification of ACID Java transactions has not been available so far. This paper provides the results on integrating a sophisticated theorem prover technology into the Java environment extended with constraints. These results include a technique based on extended Java Core Reflection for representing Java classes augmented with constraints as theories. Furthermore, techniques for enforcing behavioral compatibility and verifying transaction safety with respect to database integrity are the core results in applying the theorem prover technology to the problems of consistency of Java transactions.

2 Extending Java with Constraints and Transactions

In this section we analyze database-related issues in extending Java with a constraint language and then we present a model of Java transactions. This model is based on the interplay of declarative, logic-based constraints (specifying database integrity), bounded parametric polymorphism and orthogonal persistence.

2.1 Queries and Constraints

Complete lack of declarative, logic-based constraint capabilities in Java has major implications on incorporating queries and transactions into the Java technology. If Java had a suitably defined constraint language, such a language would naturally include (or be easily extended with) full-fledged query capabilities.

In traditional database technologies queries are functions expressed in a high-level declarative specification language. In the paradigm proposed in this paper queries are naturally objects. This reflective feature is possible because of the availability of constraints. A query is an instance of a query class given below. The invariant of this class (a constraint that applies to all visible object states outside of method execution) specifies the properties of the query result, and it thus plays the role of a qualification (filtering) expression in traditional queries.

The predefined class `Query` is both abstract and parametric. The method `qualification` is abstract. It is thus required to be implemented in a specific, user query class, by providing the actual qualification expression. An actual query method is `selectFrom`. It is applied to the argument collection of this method and the qualification expression is used for selecting the relevant elements. This condition is expressed by the invariant of this class.

```
public abstract class Query<P extends Comparable, T extends P>
{ public Query<P,T>();
  public abstract boolean qualification(T x);
  public final Collection<T> selectFrom(Collection<T> S);
  public final Collection<P> selectAndProject(Collection<T> S);
  public final OrderedCollection<P>
                       selectProjectAndOrder(Collection<T> S);
 invariant:
 Query<P,T> this; Collection<T> S; T x
 (this.selectFrom(S).contains(x) <-
                              S.contains(x), this.qualification(x),
  ...)
}
```

The form of parametric polymorphism required for proper typing of a variety of object types particularly relevant to database and persistent object systems in fact involves bounded type constraints for type parameters. This form of parametric polymorphism is required for ordered collections, dictionaries, hash tables, indices and even queries. Bounded quantification allows specification of queries that involve projection and/or ordering. The query class given above has two type parameters. T stands for the element type of the queried collection, and P for the element type of the collection that represents the result of the query. P is a subset of features of T (projection), hence the type constraint `T extends P`. The other type constraint `P extends Comparable` guarantees that the selected elements are equipped with the ordering methods.

The fact that all user-defined query classes must extend the abstract `Query` class is an explicit indication to the compiler that the assertions of these classes are subject to optimization. This is why the query methods `selectFrom`, `selectAndProject` and `selectProjectAndOrder` are final. Their implementation is entirely system-oriented and contains query optimization in particular.

The above described model for Java queries comes with some major advantages in comparison with other competitive approaches. The approach is truly object-oriented, and it allows static parsing, type checking and query optimization. By way of comparison, the ODMG model [9], although object-oriented, does not have any of the remaining advantages of the query model proposed in this project.

2.2 Persistence and Constraints

The model of persistence is based on associating persistence capabilities with the root class `Object` as illustrated below. This model is quite different from the approach for implementing orthogonal persistence adopted in PJama [4, 14] and GemstoneJ [8].

```
public class Object
{ ...
  public final Object makePersistent();
}
```

As all other classes extend `Object`, either directly or indirectly, orthogonality is guaranteed. The model of *persistence in the root* is truly object-oriented because it is based on message passing and inheritance. However, it requires re-implementation of the class `Object` and recompilation of the whole Java platform [3].

The database class given below offers features similar to the ODMG class `Database` [9] and the PJama interface `PJStore` [4] with a fundamental difference: *constraints*. The method `bind` of this class binds a name (the second argument) to an object of any type (the first argument of type `Object`). As `Class` extends `Object`, a database contains bindings for both classes and objects. The method `lookUp` returns an object (a root of persistence) of a database bound to a name. Note that the precondition and the postcondition of the method `bind` specify the semantic relationship with the method `lookUp`. A database naturally contains additional constraints.

```
public class Database
{ public Database(String name);
  public final boolean isOpen();
  public final void open()
         ensures this.isOpen();
  public final void close()
         requires this.isOpen();
  public final Object lookUp(String name)
         requires this.isOpen();
  public final boolean bind(Object x, String name)
         requires this.isOpen(), this.lookUp(name)==null,
         ensures  this.lookUp(name).equals(x);
invariant:
...
}
```

2.3 Transactions

In traditional database technologies transactions are programs. A Java model of transactions is naturally reflective: transactions are objects, instances of a class `Transaction` [9]. This view is extended in the paradigm presented in this paper by an interplay of bounded parametric polymorphism for binding a transaction to a database schema and the enforcement of database integrity constraints. Neither is available in technologies such as the ODMG Java binding [9], PJama [4, 14] or JDO [13].

A transaction is required to maintain the integrity constraints of its schema. The precondition of the method `start` requires that the integrity constraints of the transaction's schema are satisfied before the transaction begins. The postconditions of the methods `commit` and `abort` require that the integrity constraints of the transaction's schema are satisfied.

```
public abstract class Transaction <T extends Database>
{ protected    T schema;
  public       Transaction<T>(T schema);
  public final void start()
         requires schema.invariant;
  public final void commit()
         ensures schema.invariant;
  public final void abort()
         ensures schema.invariant;
  public abstract void execute()
         requires schema.invariant,
         ensures  schema.invariant;
}
```

The class `Transaction` is abstract because its method `execute` is. Specification of the actual transaction code which the `execute` method contains is thus deferred to subclasses of the `Transaction` class. However, the precondition and the postcondition of the `execute` method are specified. They require that `execute` acts as a unit of integrity with respect to the database constraints.

The appeal of static verification of a transaction is that one would guarantee by static analysis that integrity violation actually would not happen at run-time: if the integrity constraints hold before the transaction is started, they will hold at commit time. The problem is, of course, in general undecidable especially if the full fledged Java is used to write the transaction code. A sophisticated theorem prover technology is required in any case to handle the task. The other problem is caused by dynamic binding of messages to methods. This is one problem which the research on static verification of relational database transactions [19] did not have to deal with and subsequent object-oriented database research (such as [20] and [7]) has not addressed until more recently [21]. An object-oriented transaction is verified statically while the actual methods to execute the messages in the transaction code are determined only at run time. Thus if a transaction is verified, and some of the classes in its schema are subsequently extended, the transaction may violate the integrity constraints in spite of the successful static verification. This problem is solved in this paper by imposing the behavioral compatibility requirements on extending classes of a database schema. Details are given in sections 4.2, 4.3, and 4.4.

An example of a corporate database is given below. The example illustrates the primary features of the model. Usage of parametric polymorphism is shown in the collections of departments and employees, as well as in a sample transaction. A variety of constraints are given including key and referential integrity constraints in the form of invariants of the `Corporation` class, as well as preconditions and postconditions of several methods.

```
public class Corporation extends Database {
 public interface Employee{
  String name(); String ssn(); Float salary();
  Department department();
  void assignDepartment(Department d)
       ensures this.department().equals(d);
 invariant: Employee X,Y;
  X.equals(Y) <-  X.ssn().equals(Y.ssn());
 }
public interface Department {
 Integer deptNum();
 Collection<Employee> employees();
 Float allocatedPayroll();
 void addEmployee (Employee e)
      ensures this.contains(e);
 void removeEmployee (Employee e)
      requires this.contains(e);
invariant: Department X,Y;
 X.equals(Y) <- X.deptNum().equals(Y.deptNum());
 }
 public class EmployeeCollection
                      implements Collection<Employee>{
  public void add (Employee e);
 }
 public class DepartmentCollection
                      implements Collection<Department>{
  public void add (Department d);
 }

 EmployeeCollection dbEmployees;
 DepartmentCollection dbDepartments;

invariant: Employee W; Department Y;
 dbEmployees.contains(W) <- dbDepartments.contains(Y),
                            Y.employees().contains(W);
 dbDepartments.contains(Y) <- dbEmployees.contains(W),
                              W.department().equals(Y);
}

public class HireTrans extends Transaction<Corporation>{
 public HireTrans
        (Corporation corp, Employee emp, Department dept)
 {....}
 public void execute()
        requires corp.dbDepartments.contains (dept)
 {emp.assignDepartment (dept);
   dept.addEmployee (emp);
   corp.dbEmployees.add (emp);
  }
}
```

The constraints in this example are expressed in Horn clause logic. Since Corporation is a descendant of the Database class (the root of persistence), the collections dbEmployees and dbDepartments are persistent by reachability.

3 Extending the Java Virtual Machine

This section describes two extensions of the existing Java technology. The first incorporates constraints into the standard Java class files and class objects. The second allows access to these constraints via an extension of Java Core Reflection. We will consider the user interface first.

3.1 Extended Java Core Reflection

There are at least three major advantages of having constraints integrated into the Java environment and available in a declarative form:

- Checking behavioral compatibility of a subclass with respect to its superclass or a class with respect to an interface that it implements. If the constraints are compiled into code as in the existing implementations in object-oriented languages (such as Eiffel), it becomes impossible to check the behavioral subtyping conditions.

- In order to define a behavioral subtype, a user must be able to see the constraints of a supertype. This introspection cannot be based on investigating the code. In the case of interfaces the code is not available in any case and the source code of classes may not be available either. Furthermore, only a declarative form (similar to the existing form of database constraint languages) of constraints would be acceptable to database users.
- If the constraints are compiled into methods and available only in the procedural form there is no practical way of devising optimization strategies for enforcing the constraints. This is particularly important for the constraints in the classes that extend the `Database` class.

In order to make the constraints available to the users in a declarative form two extensions of the Java Core Reflection (JCR) classes are required. These extensions consist of adding new methods to the final JCR classes and hence the whole Java platform had to be recompiled in this project. However, the existing methods of the JCR classes have not been affected in any way. This is why the existing software packages that make use of JCR as it is at present should not be affected either.

The first extension allows the users of JCR to recognize the fact that parametric classes exist in this new Java environment [18, 3]. The class `Class` is thus extended with a `boolean` method `isParametric` and methods `getTypeParameters` and `getBoundTypes`.

The other extension allows introspection of constraints associated with classes and methods. The class `Class` is thus extended with a method `getDeclaredInvariant` which returns the invariant declared in the class itself. In accordance with the JCR philosophy, the method `getInvariant` returns the complete invariant which is a conjunction of the invariants inherited from the superclasses and the invariant declared in the class.

```
public final class Class {
...
 public boolean   isParametric();
 public String[]  getTypeParameters();
 public Class[]   getBoundTypes();
 public Sentence[] getInvariant();
 public Sentence[] getDeclaredInvariant()
}
```

The JCR class `Method` is extended with methods that return the method's precondition and postcondition as formulas of a particular logical system. Note that class invariants are sentences, i.e., closed formulas (all variables are quantified). The fact that the method's postcondition is a conjunction of the inherited postcondition and the declared postcondition is reflected in the method `getPostCondition`. The method's precondition is required to remain unchanged down the inheritance hierarchy, hence the method that would return the declared precondition is not available.

```
public final class Method {
...
 public Formula  getPostCondition();
 public Formula  getDeclaredPostCondition();
 public Formula  getPreCondition()
}
```

Object-oriented formulas are constructed from object-oriented terms that include constants, variables, new object constructor terms, and messages. The class `MessageTerm` contains methods that return the terms representing the receiver and the arguments of the message, as well as the method object.

```
public final class MessageTerm  extends Term {
 public Term getReceiverTerm();
 public Method getMethod();
 public Term[] getArguments();
 public Object evaluate(Object[] variables);
}
```

Atoms are in fact messages invoking `boolean` methods. In order to make this specification independent of a particular logic for expressing constraints, the classes `Formula` and `Sentence` are defined as abstract classes.

```
public abstract class Formula{
 public Variable[] getVariables();
 public abstract boolean evaluate(Object[] variables);
}
```

Since a sentence is a closed formula and contains specification of all the variables (which are universally quantified) and their types, it may be compiled independently of any context in which it may appear. This fact is reflected in the method `compile()`.

```
public abstract class Sentence extends Formula {
 public boolean compile();
}
```

Given a particular constraint language and its logic basis, specific classes representing the formulas and sentences are specified and implemented.

3.2 Constraints in Class Files

Java classes are compiled into Java class files. The problem is that this file structure was not designed with either parametric classes or assertions in mind. So the challenge is to implement both with the existing Java class file structure. Indeed, the Java platform depends heavily upon this file structure which is in fact a part of the Java Virtual Machine Specification [15].

The information about fields and methods in a class file follows the pattern given below:

```
public class MethodInfo {
 short            accessFlags;
 ConstantPoolInfo  name;
 ConstantPoolInfo  signature;
 AttributeInfo[]   attributes;
  ...
}
```

The name of the method is given as a reference to a constant pool item which contains the actual string. The signature of a method consists of types of its parameters and the result type. This type descriptor is represented as a specially formatted string. *The structure of type descriptors is determined by a grammar which is a part of the Java Virtual Machine Specification [15]. Dealing properly with the type descriptors is an essential component of the technique for implementing parametric polymorphism* presented in [3].

In the Java class file structure an array of attributes is associated with each class, each field, and each method. The structure of an attribute item is given below.

```
public class AttributeInfo {
 ConstantPoolInfo   name;
 byte[]             data;
  ...
}
```

The predefined attributes include `SourceFile`, `ConstantValue`, `Code`, `Exceptions`, `InnerClass`, `LineNumber` and `LocalVariableTable`. Out of these, the attributes `Code`, `ConstantValue` and `Exceptions` must be correctly recognized in order for the Java Virtual Machine to function correctly. An important point is that in addition to the above mentioned attributes, optional attributes are also allowed. These optional attributes have no effect on the correct functioning of the Java Virtual Machine.

Representing a schema as a Java class file is based on using optional attributes of a class file to represent inner classes similarly to the techniques reported in [3]. The information about an inner class includes its name (if it has one), references to the constant pool containing the information about the inner class itself and its outer class, and the access flags of the inner class.

In order to represent assertions in a valid Java class file structure, three additional non-standard attribute types are introduced. An additional class attribute is `Invariant`, and two additional method attributes are `PreCondition` and `PostCondition`. The structure of these attributes is determined by the chosen logic for representing assertions. The extended Java Core Reflection is based on accessing the above attributes in the loaded class objects and reporting the information on constraints in the form described in section 3.1.

As our embedding scheme relies on the optional attributes of the Java class file which are required to be ignored by the Java Virtual Machine, the first task was to modify the native class loader to store the optional attributes when loading the class file, rather than ignoring them as directed by the JVM specification. The relevant code in the Java Virtual Machine iterates over each of the class attributes in the class being loaded. Recognized attributes are handled appropriately within this loop, but unrecognized attributes are simply ignored. In the extension of this code, if an optional attribute for a constraint is found, its contents are converted from a byte array into a UTF-8 string, and a newly added pointer in the native `struct` representation of the class is set to reference the constraint string.

This change assures that the constraints are loaded from the class file and associated with the runtime class representation. Once this is accomplished, it is necessary to add methods to `Class` and `Method` objects to allow the runtime constraint representation to be accessed. This process involves modification of the native interface of the Java Virtual Machine. This extension allows calls from Java objects to interact with the internals of the JVM, in this case the runtime representation of a class. To modify the native interface, in this case adding a method to it, an entry to the array of JNINativeMethods in the native class object is added. The entry describes the name, parameters and return value of our native method called getConstraintString(). Finally, a native method is added to the JVM which simply returns the string stored in the runtime class representation.

A key implication is that when an object is promoted to persistence, its class object will be promoted to persistence as well (reachability). The `Class` object will carry all (now persistent) constraints that apply to the newly persistent object. The Java Core Reflection will report persistent constraints whenever persistent objects and their class objects are introspected.

4 Theorem Prover Technology

Incorporating a sophisticated theorem prover technology into this extended Java environment requires first of all a technique for representing Java classes extended with constraints as theories in the model-theoretic sense [1]. Methodologies for proving behavioral compatibility and transaction safety are also required, as well as their formal justification. These issues are addressed in this section.

4.1 Java Classes as Theories

The theorem prover technology used in this research is PVS [17]. PVS specifications are theories consisting of signatures of functions and the associated axioms and theorems [17]. These theories are based on a type system that includes a restricted form of semantic subtyping, parametric polymorphism, and full generality of the underlying logic basis. PVS theories are well-suited for algebraic data types but representing classes as PVS theories poses non-trivial problems. The usual thorny issues are dealing with object identities, state changes caused by mutator methods, inheritance and dynamic binding.

A parametric supporting theory ObjectRefTheory plays a key role in resolving the above issues. This theory is parameterized by the underlying object state. The abstract type ObjectRef is thus also effectively parameterized and it makes it possible to distinguish object identities from the object states that they refer to. The function state returns the object state associated with an object reference at a given time instant. The underlying temporal paradigm views the lifetime of an object as a sequence of object states, one per time instant represented by a non-negative integer.

```
ObjectRefTheory [ObjectState: TYPE+]: THEORY
BEGIN
  ObjectRef: TYPE+
  state [ObjectRef, nat -> ObjectState]
  noChange: [ObjectRef, nat -> bool]
...
noChangeAxiom: AXIOM
FORALL (ref: ObjectRef): (FORALL (n:nat):
(noChange(ref,n) IFF
               (state(ref,n) = state(ref,n+1))))
END ObjectRefTheory
```

Mutator methods are not just functions, they change the underlying object state while the object identity remains the same. In order to distinguish mutators of object state from accessor methods, the function noChange is used in the axioms to indicate that mutation of the object state does not happen. This is illustrated by the PVS DatabaseTheory in which the function lookup is an accessor and the function bind is a mutator. All of this is reflected in the complexity of the axioms in PVS theories which exceeds considerably the user's view of constraints. Note that the keyword TYPE+ denotes a nonempty type in PVS.

```
DatabaseTheory: THEORY
BEGIN
  DatabaseState : TYPE+
  IMPORTING   ObjectRefTheory[ObjectState]
  IMPORTING  ObjectRefTheory[DatabaseState]
  TYPE+ ObjectRef = ObjectRef[ObjectState]
  TYPE+ DatabaseRef = ObjectRef[DatabaseState]

 isOpen: [DatabaseState -> bool]
 open, close: [DatabaseRef -> DatabaseState]
 lookup: [DatabaseState, string -> ObjectRef]
 bind: [DatabaseState, ObjectRef, string -> DatabaseState]
 invariant: [DatabaseState -> bool]
...
lookupAxiom: AXIOM
FORALL (dr: DatabaseRef, o: ObjectRef, x: string):
  (FORALL (n: nat):
((o=lookup(state(dr,n),x)) IMPLIES
    (isOpen(state(dr,n)) AND
       noChange(dr,n) AND noChange(o,n)))
bindAxiom: AXIOM
FORALL (dr: DatabaseRef, d:DatabaseState,
                          o: ObjectRef, x: string):
  (FORALL (n: nat):
(d=bind(state(dr,n),o,x) IMPLIES
     (isOpen(state(dr,n)) AND
     (lookup(state(dr,n),x)=nil) AND
     (state(dr,n+1) = d) AND
     noChange(o,n))))
bindAndLookupAxiom: AXIOM
FORALL (dr: DatabaseRef, o: ObjectRef, x: string):
  (FORALL (n: nat):
     lookup(bind(state(dr,n), o, x),x)=o))
END DatabaseTheory
```

The availability of reflective constraints enables the automatic generation of PVS theories. A rather sophisticated Java tool, java2pvs, produces a theory for a particular class by loading the class and using the reflective methods available in class Class. Structural information required for assembling a PVS theory is available through the existing Java Core Reflection API. Java methods are represented as PVS functions, hence java2pvs examines Method objects that provide method names, parameter types, and return types. Additional semantic information expressed in PVS theories is available from our extension to Core Reflection. Class invariants are produced with a straightforward traversal of the invariant tree. Expressing preconditions and postconditions is trickier, since they are expressed in Java objects as functions on object states, and must be appropriately converted to sentences in the PVS theories. Availability of superclasses in core reflection is another necessity, since theories support the concept of inheritance via transformation of the relevant semantic properties from superclasses in theories representing subclasses.

4.2 Proving Behavioral Compatibility

In the Java type system subclasses generate subtypes, so there is no need to verify the subtyping conditions in PVS. Indeed, PVS theories are constructed from Java classes using Java Core Reflection and hence the subtyping condition is guaranteed. However, the Java type system would not know anything about behavioral subtyping as defined in [16]. In this paradigm a theory for a subclass is generated in such a way that it is necessarily behaviorally compatible with the theory of its supertype. Static verification of this condition guarantees that the results of verification are valid in spite of dynamic binding of messages to methods.

Consider a class Corporation that extends Database class. Corporation Theory contains two referential constraints expressed in the invariant axiom of this theory.

```
CorporationTheory: THEORY
BEGIN
  IMPORTING DatabaseTheory,
            EmployeeTheory, DepartmentTheory,
  IMPORTING CollectionTheory[EmployeeState],
            CollectionTheory[DepartmentState]
 dbDepartments: VAR CollectionRef[DepartmentState]
 dbEmployees: VAR CollectionRef[EmployeeState]
%Transformed signatures and axioms from DatabaseTheory
 invariantAxiom: AXIOM
FORALL (dr: DepartmentRef, er: EmployeeRef):
  FORALL (n:nat):
((contains(state(dbDepartments,n),dr) AND
  contains(state(employees(dr),n),er)) IMPLIES
      contains(state(dbEmployees,n),er))) AND
FORALL (dr: DepartmentRef, er: EmployeeRef):
  (FORALL (n:nat):
((contains(state(dbEmployees,n),er) AND
  (dr=department(state(er,n)))) IMPLIES
      contains(state(dbDepartments,n),dr)))
END CorporationTheory
```

There are two properties that we would like to hold in order for the objects of the class Corporation to be behaviorally compatible with the objects of the class Database. The first one is the inheritance of method signatures and the associated constraints. This is why CorporationTheory includes the signatures and axioms of the DatabaseTheory that are appropriately translated in the process of constructing CorporationTheory using Java Core Reflection. In this process the superclass Database of the class Corporation is accessed to inspect the signatures of methods and the associated constraints. For example, the translated accessor lookup and the mutator bind and the axiom that relates them will be included in CorporationTheory as follows:

```
lookup: [CorporationState, string -> ObjectRef]
bind:  [CorporationState, ObjectState, string -> CorporationState]
bindAndLookupAxiom: AXIOM
FORALL (dr: CorporationRef, o: ObjectRef, x: string):
  (FORALL (n: nat):
(lookup(bind(state(dr,n), o, x),x) = o))
```

This guarantees that all the axioms of the DatabaseTheory hold in CorporationTheory. Of course, this transformation of method signatures does not follow the rules of subtyping, but this fact has no negative implications since the rules of subtyping are enforced by the Java type system and PVS theories are used just to verify the behavioral properties which are not expressible in a type system.

The other part of the behavioral compatibility requirement applies to substitutability. Objects of type Corporation should be substitutable for objects of type Database with no behavioral discrepancies. This condition is expressed as a theorem of CorporationTheory. *In specifying this theorem the fact that PVS is based on higher order logic is critical.* This theorem requires the existence of the abstraction function

```
absF: [CorporationRef -> DatabaseRef]
```

so that all the axioms of DatabaseTheory hold when every object x of type Database is replaced with an object absF(y) where y is an object of type Corporation. This theorem which includes only the bindAndLookup axiom is specified below:

```
bindAndLookupTheorem: THEOREM
EXISTS (absF: [CorporationRef -> DatabaseRef]):
(FORALL (dr: CorporationRef, o: ObjectRef, x: string):
(FORALL (n: nat):
 (lookup(bind(state(absF(dr),n), o, x),x) = o)))
```

The complete theorem would include all the remaining axioms of DatabaseTheory. Verification of this theorem in CorporationTheory clearly requires the availability of DatabaseTheory which is thus necessarily imported in CorporationTheory.

4.3 Model Theoretic Justification

The approach to verification of behavioral compatibility from section 4.2 has the following model theoretic justification based on the results presented in [1].

A theory $Th_A = (\Sigma_A, E_A)$ consists of a collection of function signatures Σ_A and a finite set E_A of Σ_A sentences (closed formulas).

A Σ_A model specifies a function for each function signature in Σ_A. A collection of Σ_A models is denoted as $Mod(\Sigma_A)$.

If $Th_A = (\Sigma_A, E_A)$ is a theory then a model M_A for Th_A is a Σ_A model which satisfies the following condition:

If $e \in E_A$ is a sentence with variables X then for any substitution of variables $f : X \to M_A$, $e < f(X) >$ evaluates to true where $e < f(X) >$ denotes the formula e in which the quantifiers are removed and a substitution of variables X is performed according to f.

If the above conditions are satisfied we say that M_A satisfies all sentences of E_A, denoted as:
$M_A \models e$ for all $e \in E_A$.

If $Th_A = (\Sigma_A, E_A)$ and $Th_B = (\Sigma_B, E_B)$ are theories,
$F : Th_A \to Th_B$ is a theory morphism iff
$F : \Sigma_A \to \Sigma_B$ is a mapping of function signatures such that
$M_B \models e$ for all $e \in E_B \Rightarrow M_B \models F(e)$ for each $e \in E_A$.

It is easy to prove that the construction of the theory for a subclass B of a class A presented in section 4.2 has the following property.

Lemma (*Theories for subclasses*)
If A is a class and B is its subclass, then Th_B is constructed from Th_A in such a way that there exists a theory morphism $F : Th_A \to Th_B$.

The generic form of the theorem whose proof is attempted by the PVS theorem prover is:

Theorem (*Behavioral compatibility*)

A theory $Th_B = (\Sigma_B, E_B)$ is behaviorally compatible with a theory $Th_A = (\Sigma_A, E_A)$ if:

Given a theory morphism $F : Th_A \to Th_B$ there exists an abstraction function

$absF : Mod(\Sigma_B) \to Mod(\Sigma_A)$ such that
$absF(M_B) \models e$ for all $e \in E_A$.

Note that $M_B \models F(e)$ for all $e \in E_A$ is already guaranteed by the fact that F is a theory morphism.

In the above theorem $absF(M_B) \models e$ for all $e \in E_A$ means that for any substitution of variables $f : X \to absF(M_B)$, $e < f(X) >$ evaluates to true.

4.4 Transaction Verification

The above approach applies directly to the problem of verification of database transactions. An incomplete parametric transaction theory has the following form:

```
TransactionTheory[Database: TYPE+]
BEGIN
  IMPORTING  DatabaseTheory
 execute: [DatabaseState -> DatabaseState]
..
IntegrityAxiom: AXIOM
FORALL(dr: DatabaseRef, d: DatabaseState):
  (FORALL (n:nat):
((invariant(state(dr,n)) AND
   (d=execute(state(dr,n)))) IMPLIES
       ((state(dr,n+1)=d) AND invariant(d))))
END TransactionTheory
```

In generating the theory `TransactionTheory[Corporation]` for an instantiated class `Transaction<Corporation>` the type checker verifies that `Corporation extends Database`. In addition the signatures of the predicate `invariant` and the function `execute` are transformed into:

```
invariant: [CorporationState -> bool]
execute: [CorporationState -> CorporationState]
```

The integrity axiom is transformed into the following form where the definition of the predicate `invariant` given in `CorporationTheory` applies:

```
IntegrityAxiom: AXIOM
FORALL(dr: CorporationRef, d: CorporationState):
  (FORALL (n:nat):
((invariant(state(dr,n)) AND
            (d=execute(state(dr,n))))
  IMPLIES ((state(dr,n+1)=d) AND invariant(d))))
```

The compromise between static and dynamic checking amounts to treating constraints that are meant to be checked at run-time as axioms and proving the remaining ones as theorems. The proofs will be valid as long as the truth of the axioms is guaranteed at run-time. At the same time the constraints that are proved as theorems under the above assumptions will not be checked at run-time. In particular, if method-related constraints are verified at run-time and database invariants are proved as theorems, extensive effort in verifying database invariants at run-time will be avoided. In addition, verification does not have to deal with procedural code at all as its behavior will be checked dynamically, which makes the task much more tractable.

5 Related Work

The most important language binding of the ODMG Standard [9] is the Java binding. However, it is completely lacking logic-based integrity constraints (some constraints are specified in separate property files in a very awkward fashion). This is in contradistinction to earlier object-oriented database research on constraints such as [10, 7].

PJama [4, 14] (an orthogonally persistent Java system) is lacking any form of declarative constraint capabilities and hence the notion of a transaction in PJama lacks a fundamental component: maintaining integrity constraints. Other related and unimplemented models related to PJama are [5, 6]. But not one of these results guarantees database integrity with respect to a set of declarative (possibly logic-based) constraints that are specified as a critical component of the data model.

Java Data Objects (JDO) [13] is the most important follow-up work to the ODMG Standard. There is no way to specify logic-based integrity constraints so that the notion of a transaction is impaired just as in the case of PJama or ODMG.

Earlier research includes a particularly important result [19] that makes use of computational logic and applies to the relational model. Two projects on the usage of theorem prover technologies that are particularly relevant are [11] and [21]. Research reported in [11] is directed toward Java and it is based on the PVS theorem prover. This is mainly a reasoning tool used for analyzing the procedural code but also to verify class invariants. Verification of behavioral subtyping is not reported in [11]. Research reported in [21] is directed toward object-oriented databases and it is based on the Isabelle/HOL theorem prover. In the earlier research [20] the ODMG model is extended with constraints expressed in the OQL style equipped with transactions that maintain the integrity constraints (unlike ODMG). The more recent work captures nontrivial subtleties of the object-oriented paradigm (such as dynamic binding) in a generic theory of objects which in fact includes persistence.

This paper establishes the results that relate the problems of behavioral compatibility as expressed by the notion of behavioral subtyping [16] with transaction safety. The core of the approach in using the PVS theorem prover technology is based on the model-theoretic results about behavioral compatibility for the object-oriented paradigm established in [1] and [2].

6 Conclusions

Without database integrity constraints there is no way of introducing database transactions into Java. But in fact, a sophisticated model of Java transactions requires an interplay of bounded parametric polymorphism and constraints. More importantly, two main results are required in order to be able to verify database integrity with respect to Java transactions. One of them provides a solution for integrating logic-based constraints into the existing Java technology. The other is the integration with a sophisticated theorem prover technology. The specific contributions of this paper are:

- A model of Java transactions based on the interplay of logic-based constraints, bounded parametric polymorphism and inheritance.
- A technique for integrating logic-based constraints and parametric classes into the existing Java environment, Java class files and class objects in particular, along with the required extension of Java Core Reflection to report correctly information about constraints and parametric classes.
- A model for enforcing database integrity constraints that relies on a sophisticated theorem prover technology and combines static and dynamic techniques to attain reasonable performance in constraint checking.

Acknowledgment

The authors would like to thank Brian Cabana and Brent Atkinson for their collaboration in this project.

References

1. S. Alagić, S. Kouznetsova, Behavioral compatibility of self-typed theories. Proceedings of ECOOP 2002, *Lecture Notes in Computer Science 2374*, pp. 585-608, Springer, 2002.
2. S. Alagić, Semantics of temporal classes, *Information and Computation*, *163*, pp. 60 - 102, 2000.
3. S. Alagić and T. Ngyen, Parametric polymorphism and orthogonal persistence, Proceedings of the ECOOP 2000 Symposium on Objects and Databases, *Lecture Notes in Computer Science 1813*, pp. 32 - 46, 2001.

4. M. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence, An orthogonally persistent Java^{TM}, ACM SIGMOD Record *25*, pp. 68-75, ACM, 1996.
5. L. Daynes, Extensible transaction management in PJava in: R. Morrison, M. Jordan, and M. Atkinson: *Advances in Persistent Object Systems*, Morgan Kaufmann Publishers, 1999.
6. L. Daynes, M. Atkinson and P. Valduirez, Efficient support for customized concurrency control in Persistent Java, Proceedings of the Int. Workshop on Advanced Transaction Models and Architecture, (in conjunction with VLDB 96), India, Sept. 1996.
7. V. Benzaken and D. Doucet, Themis: A database language handling integrity constraints, *VLDB Journal 4*, pp. 493-517, 1994.
8. B. Bretl, A. Otis, M. San Soucie, B. Schuchardt, and R. Venkatesh, Persistent Java objects in 3 tier architectures, in: R. Morrison, M. Jordan, and M. Atkinson: *Advances in Persistent Object Systems*, pp. 236-249, Morgan Kaufmann Publishers, 1999.
9. R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, 2000.
10. N. Gehani and H. V. Jagadish, Ode as active database: Constraints and triggers, Proceedings of the VLDB Conference, pp. 327-336, Morgan Kaufmann, 1991.
11. B. Jacob, J. de Berg, M. Husiman, and M. van Berkum, Reasoning about Java classes, Proceedings of OOPSLA '98, pp. 329 -340, ACM, 1998.
12. Java Core Reflection, JDK 1.1, Sun Microsystems, 1997.
13. Java^{TM} Data Objects, JSR 000012, Forte Tools, Sun Microsystems Inc., 2000.
14. M. Jordan and M. Atkinson, Orthogonal persistence for Java - A mid-term report, in: R. Morrison, M. Jordan, and M. Atkinson: *Advances in Persistent Object Systems*, pp. 335 - 352, Morgan Kaufmann Publishers, 1999.
15. T. Lindholm and F. Yellin, *The Java^{TM} Virtual Machine Language Specification*, Addison-Wesley, 1996.
16. B. Liskov and J. M. Wing, A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems 16*, pp. 1811-1841, 1994.
17. S. Owre, N. Shakar, J. M. Rushby, and D. W. J. Stringer-Clavert: PVS Language Reference, SRI International, Computer Science Laboratory, Menlo Park, California.
18. J. Solorzano and S. Alagić, Parametric polymorphism for Java^{TM}: A reflective solution, Proceedings of OOPSLA '98, pp. 216-225, ACM, 1998.
19. T. Sheard and D. Stemple, Automatic verification of database transaction safety, *ACM Transactions on Database Systems 14*, pp. 322-368, 1989.
20. D. Spelt and H. Balsters, Automatic verification of transactions on an object-oriented database, in: S. Cluet and R. Hull (Eds.), Database Programming Languages, *Lecture Notes in Computer Science 1369*, pp. 396-412, 1998.
21. D. Spelt and S. Even, A theorem prover-based analysis tool for object-oriented databases, in: Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science 1579*, pp 375 - 389, Springer, 1999.

Integrating Database and Programming Language Constraints

Oded Shmueli[*], Mukund Raghavachari, Vivek Sarkar, Rajesh Bordawekar, and Michael G. Burke

IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA

Abstract. We examine the maintenance of data consistency in the presence of application-database interactions. Currently, a programmer must insert explicit checks to ensure that data are consistent with respect to the application's and the database's requirements. This explicit hand-coding of integrity constraint checks is error-prone and results in less maintainable code when the integrity constraints on the application or the database change. We describe an architecture for automatically generating application-level checks from application and database integrity constraint specifications. We focus on EJB-database interactions as a concrete context for our work, but our results are applicable to other programming language-database interactions.

1 Introduction

Programming models for accessing databases have evolved significantly from batch systems and transaction processing monitors for procedural applications to container-based application servers and web services for distributed object-oriented applications. A theme in this progression is a shift in the boundary between applications and database systems. Recent programming models such as *web services* treat accesses to remote databases in much the same manner as accesses to remote applications — the database is yet another service in a distributed environment.

The early programming models were database-centric, and the database was responsible for robustness guarantees such as the ACID (Atomicity, Consistency, Isolation, and Durability) properties of transactions. The blurring of the boundary between databases and applications necessitates the development of new programming language abstractions for managing data access in an integrated application-database context. First-class application-level support for functionalities such as transactions, data consistency, logging, etc. are necessary to simplify the development of web services applications.

The focus of this paper is the *maintenance of data consistency in an integrated application-database environment.* Currently, the application programmer must insert explicit checks to ensure that data are consistent with respect to the

[*] Author's Current Address: Computer Science Department, Technion – Israel Institute of Technology, Haifa, Israel.

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 90–109, 2004.

application's and the database's requirements. This explicit hand-coding of integrity constraint checks is error-prone and results in less maintainable code when the integrity constraints on the application or the database change. Furthermore, when the database enforces data constraints in deferred mode, the inconsistency of application data with respect to the database may not be identified until the end of a transaction, which may result in significant loss of work due to rollbacks. This is also the case for *disconnected* applications.

Our approach is to use *declarative specifications* of integrity constraints for application models and database schemas, and a combination of *static and dynamic checking* to ensure that an application is correct with respect to these specifications. There are three kinds of checking being pursued in our approach. First, *dynamic checks* inserted into application code (automatically or programmer-directed) verify at runtime that the database-level and application-level integrity constraints are satisfied. The failure of a runtime check raises an exception which can be caught by the application to compensate for the violation. This contrasts with the current situation, where a transaction may rollback in a database at commit time due to a violation of a database integrity constraint, without raising a proper exception at the application level (since the database is unaware of the internals of the application). Second, *static co-analysis of application and database models* can be performed to identify inconsistencies between application constraints and database constraints at the model level. Finally, *static analysis* of the application code can be used to determine at development time if any constraints are definitely violated, and also to simplify or even eliminate runtime checks. This paper focuses on the first kind of checks, dynamic checks. The other classes of checks are being pursued as follow-on research to this work in the context of the DOMO (Data-Object Modeling and Optimization) project at IBM Research.

Our approach offers several advantages. When integrity checks are automatically inserted by the system from declarative specifications, it is easier to handle changes in integrity requirements than in the case where these checks are embedded in application code by hand. Also, automatic verification of data integrity constraints enables applications to detect and compensate for integrity violations by handling exceptions. Finally, the declarative specifications can enable integrity constraints to be checked by applications in *deferred mode* and even in *disconnected mode*, thereby improving performance by avoiding the need to be connected to the database at all times. Our contention is that these advantages result in more robust and maintainable applications, especially in distributed application scenarios like web services.

Enforcement of integrity constraints at the application level raises interesting technical challenges. At any point in time, some application data correspond to a subset (containing tuples' projections) of tuples in the underlying database. One may be unable to determine conclusively, at the application level, whether an integrity constraint on the underlying database will be violated. We shall, in these cases, generate *approximate* application-level checks. The failure of an approximate check signals that the application's data are indeed inconsistent with respect to the database integrity constraints. The success of an approximate check is inconclusive: the constraint must eventually be checked in the database. Approximate checks are especially relevant when the database is not available, for example, when performing static analysis of application code to detect integrity violations or when the application operates in disconnected mode.

To provide a concrete context for our work, we explain our approach in terms of interactions between EJBs (Enterprise Java Beans) [19] and relational databases. However, our techniques are applicable to other models of application-database interactions, such as ADO [18] and JDO [17]. The EJB specification simplifies the integration of an application with a database by providing Container-Managed Persistence (CMP); a developer defines EJBs with respect to an *abstract persistence schema*, and then, specifies a *deployment mapping* between the abstract persistence schema and a database. At specific points during the execution of an EJB, the runtime system synchronizes the state of the EJB with values in the database, updating the EJB or the database as necessary.

In the EJB CMP model, the application developer bears the burden of reasoning about the correctness of the application with respect to database integrity constraints. The programmer must insert explicit checks to ensure that updates made to application objects will not result in integrity violations and/or rollbacks in the database. Our approach provides an architecture for coordinating integrity constraint checks between EJB application code and the database(s), as well as, a common framework for reasoning about application and database integrity constraint specifications. In the context of EJBs, we explain how to maintain and deduce information at the application level using a *shadow database*. The shadow database provides local information for supporting dynamic constraint checking. We detail how checking code is generated and inserted at important points in an EJB life-cycle.

The contributions of this paper are as follows:

1. An architecture designed for dynamically monitoring application-database interactions and detecting violations in application-level code, based on a novel mechanism called the shadow database.
2. A formalism for reasoning about application and database integrity constraints. Our formalism supports the representation of string and numeric domain constraints, referential integrity constraints, as well as mappings between EJBs and database relations.
3. Techniques for analyzing constraints and generating application-level checks from them. Key to our approach is the notion of approximate checks (including *formula expansion*), which allows us, in practice, to catch common integrity violations without accessing the database.
4. A detailed illustration of how the above techniques can be applied to the concrete context of interactions between EJBs and databases.

Example 1. Consider the sample relations and integrity constraints shown in Figure 1. Let `EmployeeEJB` be an EJB class with fields {`NAME`, `DEPT`, `MGRID`, `SALARY`} that are mapped to the corresponding columns of the `EMPLOYEE` relation. As an example of dynamic checking, given an instance of `EmployeeEJB`, we can detect violations of C1 and C2 by inserting dynamic checks of values local to the EJB instance at appropriate commit points in the EJB's execution. As an example of static checking, note that dynamic checks for constraint C1 on EJBs are equivalent to *null pointer checks* on Java object references at the commit points. Standard compiler optimization techniques, such as partial redundancy elimination [16], can be used to eliminate or simplify these dynamic checks.

For a more complex example, consider an application that sets the `DEPT` field to "USSales" and the `SALARY` field to 200,001. Static checking can detect that this set of values will raise an integrity violation of C5. C5 states that if an employee is in the "USSales" department, then the employee's salary must be less than the employee's manager's salary. C9, however, states that a manager's salary cannot be greater than 200,000. This example reveals the importance of examining the interactions among multiple constraints. Though data about `MANAGER` tuples are not present in the EJB class, we can infer facts about these tuples that aid in determining violations of integrity constraints. Past work on checking of inequality constraints in optimizing compilers (e.g., for data dependence analysis [1] or array bounds checks [5]) provide a foundation for this level of static checking.

As mentioned earlier, not all database integrity checks can be checked at the application level. This is because an instance of an EJB class contains only a projection of the corresponding tuple in the database, For example, the `EmployeeEJB` class does not contain a field corresponding to the `BONUS` column. As a result, we cannot dynamically or statically detect violations of constraint C3 at the application level in the standard EJB CMP model[1].

Related Work. Bernstein *et al.* [3, 2] investigate methods for efficiently enforcing semantic integrity assertions. Constraint representation and maintenance in the context of object-oriented databases has been investigated by Jagadish and Qian [13]. This paper also investigates maintenance of inter-object constraints. Maintaining consistency of replicated data across loosely-coupled, heterogeneous databases has been examined by Grefen and Widom [8].

A problem similar to the one we treat is that of integrity maintenance and consistency in the presence of limited information. One setting in which such problems arise naturally is distributed and federated databases. Techniques for verifying global integrity constraints are presented by Gupta and Widom [10]. Using global constraints and local data as input, their algorithm generates a local condition, which, when satisfied, guarantees global constraint satisfaction. Gupta *et al.* [9] seek a complete test that can determine conclusively, using only accesses to local data, that an update *will not* cause a violation of global constraints. In contrast, we desire useful and tractable tests that determine conclusively that an update *will* cause a violation of global constraints. Our tests handle referential

[1] It may be possible to define an extension of the EJB CMP model that implicitly extends an EJB class to include fields necessary for complete checking of integrity constraints, but such an extension could lead to additional complexities and performance overheads.

EMPLOYEE

NAME	DEPT	MGRID	SALARY	BONUS

(C1) NAME $\neq$ NULL
(C2) DEPT = "US$*$"
(C3) BONUS $\leq 0.4 \times$ SALARY
(C4) FOREIGN KEY (DEPT, MGRID) REFERENCES MANAGER(DEPT, MGRID)
(C5) (DEPT = "USSales") $\Rightarrow$ SALARY $\leq$ MGR(DEPT, MGRID).SALARY

MANAGER

NAME	DEPT	MGRID	SALARY

(C6) NAME $\neq$ NULL
(C7) DEPT = "US$*$"
(C8) $500 \leq$ MGRID ≤ 999
(C9) $100000 \leq$ SALARY ≤ 200000

Fig. 1. Sample relations in a database with associated integrity constraints. We assume that (DEPT, MGRID) in the EMPLOYEE relation is a foreign key to the MANAGER relation.

integrity by creating a local approximation of remote data; we do not have a complete local database, but a partial view into one. Our approximation approach, and disallowing explicit negation, are designed to facilitate the use of (integer/linear) solvers at compile time (and more tractable approximations at run-time).

Brucker and Wolff [6] have proposed an approach that uses UML class diagrams annotated with Object Constraint Language (OCL) constraints to generate code automatically for runtime checking of distributed components developed using EJBs. The UML class diagrams encode preconditions, postconditions, and class invariants associated with the components. Our architecture, however, is more general in that it is capable of encoding and reasoning about constraints on EJBs *and relational databases.*

Structure of the Paper. Section 2 presents background on data integrity constraints and Enterprise Java Beans. Section 3 describes our formalism for representing integrity constraints. Section 4 details how application-level checks that detect integrity violations are generated, and we conclude in Section 5.

2 Background

2.1 Database Integrity Constraints

Databases provide many mechanisms for the specification of constraints [12, 14]. One may set constraints on a column of a relation, between columns of a relation, or between relations. The data type of a column is an elementary constraint that limits the set of allowed content values. Domain constraints can further constrain the values of tuples. A referential integrity constraint states that certain column values in one relation (the child relation) must be a key, or unique, in another relation (the parent relation). Referential integrity constraints are a powerful mechanism for encoding relationships and constraints on their multiplicities, such as 1-1 and 1-n. Foreign keys are columns in one (child) relation that refer to columns in another (parent) relation such that the combination of columns at the parent is declared as either unique or as a primary key. In specifying a foreign key, the database designer has the option of specifying what happens if a parent relation tuple is deleted (or updated) while being pointed to by foreign key references from other relations. The options are to block the deletion, to cascade it (to delete or update the referring tuples), to set the referring columns to `NULL` or to set them to a default value.

2.2 Enterprise Java Beans

Enterprise Java Beans (EJB) Technology is part of the Java 2 Enterprise Edition specification. An EJB is a collection of Java classes defining a set of server-side classes. Instances of an EJB are objects (encapsulating data and methods) that reside on the server and are accessed, possibly remotely, by a client. We use the terms *bean* and *bean instance* to refer to the various facets of an EJB.

We focus on *entity* beans that use *Container-Managed Persistence* (CMP). An entity bean instance represents a database-derived object. With container-managed persistence, the application developer provides a declarative specification in a file called the *deployment descriptor* of the mapping from fields of a bean to columns of a relation. A subset of the fields of an entity bean is designated to be the key for the entity bean. The columns in a relation corresponding to these fields must form a key in the relation. Once the key has been set for an entity bean instance, it may not be changed. The developer may also specify *Container-Managed Relationships* (CMR) between entity bean classes, which imply foreign key constraints between the relations corresponding to the entity beans.

A J2EE-compliant application server processes the declarative specification provided by the developer to generate code that manages database interactions. The runtime system on which entity bean instances execute, called the *container*, manages the container-managed relationships, concurrency, loading, and storing of data from the database. We now describe a simplified model of EJB-database interactions, which corresponds to common uses of EJBs.

An application may either work with entity bean instances populated with values from the database or create new entity bean instances and insert the corresponding data into the database. Interactions with the database normally occur within the scope of an entity bean transaction, which we assume to map directly to a database transaction. All updates to the database are committed (resp., rolled back) when the entity bean transaction commits (resp., rolls back). We assume that the scope of a transaction is limited to a single container (i.e., no distributed transactions across containers). There are three kinds of interactions of interest:

- `EJBLoad`: An instance of an entity bean is created by loading a tuple from the database.
- `EJBStore`: The tuple in the database with the same key as the entity bean instance is updated with the values of the entity bean. If no such tuple exists, a new tuple is inserted.
- `EJBRemove`: Remove the unique tuple in the database corresponding to the entity bean instance being removed, using a `DELETE` statement. The tuple is identified by the EJB key.

3 Representing Constraints

Our representation of constraints encodes database and application constraints and the mapping between application objects and database relations. The representation has been designed to be amenable to analysis by a constraint solver. We detail the expression of constraints, the expansion of constraint formulas with respect to other formulas, and the generation of constraint formulas from relational and application-level specifications.

3.1 Expressing Constraints

We represent a relation in a database or an EJB class as an entity, $e(\overline{X})$, where $\overline{X} = \{X_1, \ldots, X_n\}$ is an ordered set of variables that correspond to columns in a relation or fields of an EJB. We illustrate our constraints in terms of relations and tuples; the extension to entity beans is straightforward. For each entity, $e(\overline{X})$, there is a constraint formula, C, of the form, $e(\overline{X}) \Rightarrow$ (*vars*, *unique*, *refs*, *dom*), where:

- *vars*: Set of disjoint ordered sets of variables, $\{\overline{Y_1}, \overline{Y_2}, \ldots, \overline{Y_k}\}$.
- *unique*: Set of elements of the form **unique**$(ent, \overline{Z}, \overline{Z'})$, where *ent* is an entity, possibly e, $\overline{Z'} \subseteq \overline{Z}$, and $\overline{Z} \in vars \;\cup\; \{\overline{X}\}$.
- *refs*: Set of elements of the form $e'(\overline{Y}) \wedge$ **agree** $((\overline{X'}), (\overline{Y'}))$, where $\overline{X'} \subseteq \overline{X}, \overline{Y'} \subseteq \overline{Y}, \overline{Y} \in vars.$
- *dom*: Quantifier-free first-order predicate logic formula built using *type declarations*, *string constraints*, *non-null constraints*, and *numeric constraints*, as will be described.

The set *vars* contains disjoint sets of variables. The set *unique* represents declarations of uniqueness constraints on columns of entities. The set, *refs*, describes referential integrity constraints between entities; such a constraint states that the existence of $e(\overline{X})$ implies the existence of $e'(\overline{Y})$ with pair-wise equality between $(\overline{X'})$ and $(\overline{Y'})$. Finally, *dom* is a (quantifier-free) first-order predicate logic formula in disjunctive normal form (DNF), built using the logical connectives $\wedge$ and $\vee$. We do not allow negations in order to facilitate feasibility checking. The semantics is summarized in Appendix A.

Given a variable, $Z \in \overline{X}$ or $Z \in \overline{Y}, \overline{Y} \in vars$, a *type declaration* is of the form **integer**(Z), **string**(Z), **float**(Z), **not-string**(Z), **not-float**(Z), or **not-integer**(Z). A *string constraint* is of the form **matchstring**($Z, expr, [a, b]$). where *expr* is a regular expression [11]. The interval, $[a, b]$, bounds the length of the string. a is an integer constant and b is either an integer constant greater than or equal to a, or ∞ ($b = \infty$ implies that the string may be arbitrarily long). For readability, we use $Z = expr$ to denote **matchstring**($Z, expr, [0, \infty]$) and $Z \neq expr$ to denote **matchstring**($Z, \overline{expr}, [0, \infty]$), where $\overline{expr}$ is the complement of *expr*.

A *non-null* constraint is of the form $Z \neq$ `NULL`, where `NULL` is a distinguished constant. A numeric constraint is an arithmetic relationship operator from $\{<, >, \geq, \leq, =, \neq\}$ relating two expressions built using $\{+, -, *, /\}$, variables from $\overline{X}$ or $\overline{Y} \in vars$, and integer and float constants.

Example 2. Consider the sample relations of Figure 1. Let $\overline{X} = \{X_1, \ldots, X_5\}$ and $\overline{Y} = \{Y_1, \ldots, Y_4\}$. The constraint formulas for the two entities, `EMPLOYEE`, and `MANAGER` are[2]:

`EMPLOYEE`($\overline{X}$)$\Rightarrow$
vars : $\{\overline{Y}\}$
unique : $\emptyset$
refs : $\{$`MANAGER`$(\overline{Y}) \wedge$ **agree**$((X_2, X_3), (Y_2, Y_3))\}$
dom : **string**(X_1)$\wedge$**string**(X_2) $\wedge$ **integer**(X_3)$\wedge$ `MANAGER`($\overline{Y}$)$\Rightarrow$
integer(X_4) $\wedge$ **integer**(X_5) $\wedge$
$(X_1 \neq$ `NULL`$) \wedge (X_2 =$ "US*"$) \wedge (X_5 \leq$
$0.4 \times X_4) \wedge (X_2 \neq$ "USSales"$\vee (X_4 \leq Y_4))$

vars : $\emptyset$
unique : { **unique**(`MANAGER`, $\overline{Y}, \{Y_2, Y_3\}$) }
refs : $\emptyset$
dom : **string**(Y_1) $\wedge$ **string**(Y_2)$\wedge$ **integer**(Y_3) $\wedge$
integer(Y_4)$\wedge$ $(Y_1 \neq$ `NULL`$) \wedge (Y_2 =$
"US*"$)\wedge (Y_3 \geq 500) \wedge (Y_3 \leq 999) \wedge (Y_4 \geq$
$100000) \wedge (Y_4 \leq 200000)$

3.2 Expansion of Constraints

Constraint formulas may contain references to entities that have constraint formulas themselves, with the chain of references perhaps being recursive. It is often useful to expand the constraint formula for an entity. Through expansion, we can examine the interactions among the constraints of different entities. Expansion can increase the precision of our checks, as will be illustrated in Example 3. In case of recursion in the references of constraint formulas, we will bound the depth of expansion performed. Conceptually, we view an expanded formula as a tree, with edges corresponding to *references*. The expansion of a formula is then performed in a bottom-up manner (from leaf to root) on the tree.

[2] For readability, we do not show these formulas in disjunctive normal form.

Let $C : (\overline{X}) \Rightarrow$ (*vars*, *unique*, *refs*, *dom*), where $e'(\overline{Y}) \wedge \mathbf{agree}((\overline{X'}), (\overline{Y'})) \in$ *refs* be a constraint formula. If e' has a constraint formula, $C' : e'(\overline{Z}) \Rightarrow$ (*vars'*, *unique'*, *refs'*, *dom'*), the expansion of C with respect to C' is the formula:

$e(\overline{X}) \Rightarrow$ (*vars* $\cup$ *vars'* $\cup$ $\overline{Z}$, *unique* $\cup$ *unique'*,
refs, *dom* $\wedge$ *dom'* $\wedge$ *equality constraints*)

where *equality constraints* denotes equality constraints between the variables from $\overline{Y}$ and the corresponding variables in $\overline{Z}$ [3]. We assume that either *vars*, $\overline{Z}$ and *vars'* are disjoint, or *vars'* and C' are renamed appropriately. Note that, in general, C' may itself be an expanded formula. The expression, *dom* $\wedge$ *dom'* $\wedge$ *equality constraints* is converted into disjunctive normal form. We shall often simplify formulas by propagating equalities implied by the *equality constraints* and by **agree** clauses of the formula.

Example 3. Let e be an entity with columns A and B and f an entity with columns C and D. Consider the constraint $e(\{A, B\}) \Rightarrow$ ({{C, D}}, $\emptyset$, *refs*, *dom*), where:

refs : $\{f(C, D) \wedge \mathbf{agree}((B), (C))\}$
dom : $\mathbf{integer}(A) \wedge \mathbf{integer}(B) \wedge (D > A + 2)$

Consider an update, $A = 7$, to an instance of entity e. If we consider the feasibility of *dom*, with respect to setting $A = 7$, we find that is satisfiable. It is possible that there exists some entity that satisfies $f(C, D)$, where D is greater than $9 = 7 + 2$.

Let the constraint formula for f be $f(\{M, N\}) \Rightarrow$ ($\emptyset$, { $\mathbf{unique}(f, \{M, N\}$,$\{M\})$ }, $\emptyset$, *dom'*), where:

dom' : $\mathbf{integer}(M) \wedge \mathbf{integer}(N) \wedge (N < 10)$

If we expand the formula for e by factoring in information about f, we get an expanded formula:

vars : $\{\{C, D\}, \{M, N\}\}$
unique : $\{$ $\mathbf{unique}(f, \{M, N\}, M)$ $\}$
refs : $\{f(C, D) \wedge \mathbf{agree}((B), (C))\}$
dom : $\mathbf{integer}(A) \wedge \mathbf{integer}(B) \wedge (D > A + 2) \wedge (C = M) \wedge (D = N) \wedge \mathbf{integer}(M) \wedge \mathbf{integer}(N) \wedge (N < 10)$

Propagating the equalities, $(C = M)$ and $(D = N)$, and the equality implied by the **agree** constraint, $(B = C)$, and removing redundant constraints, we get a simplified *dom* formula:

$$\mathbf{integer}(A) \wedge \mathbf{integer}(B) \wedge \mathbf{integer}(D) \wedge$$
$$(D > A + 2) \wedge (D < 10)$$

The expression, *dom*, is no longer satisfiable when $A = 7$. Since $D > A + 2$, $D > 9$. We also have $D < 10$. Therefore, $9 < D < 10$, which is infeasible since D must be an **integer**. Note that expansion allows us to reason about entities, such as f, without fetching them from the database. Further, if it is possible to

[3] Equality constraints are a mechanism for associating variables in one entity to those in another.

statically determine that the value of A computed by a given program is greater than 7, then static checking can be used to determine that the program violates the constraint formula.

3.3 Populating Constraints

The EJB standard supports the specification of referential integrity constraints, which may be encoded within the *refs* components of formulas in our formalism. There is no mechanism for specifying domain constraints on entity beans corresponding to the *dom* components of our formulas. Our representation maps closely to a subset of OCL [20], which is one possible means of specifying application-level constraints. We assume that the deployment descriptor allows specification of domain formulas — the exact syntax for specifying these formulas is omitted.

For each EJB in the deployment descriptor of an application, the mapping between an entity bean and a relation is viewed as a referential integrity constraint. For the `EmployeeEJB` bean of Example 1, let $e(\overline{X})$ represent the corresponding entity with a constraint formula C. The *refs* component of C will contain a reference $\texttt{EMPLOYEE}(\overline{Y}) \wedge \mathbf{agree}(\overline{X}, \overline{Y'})$ that represents the mapping of the `EmployeeEJB` bean to the `EMPLOYEE` relation of Figure 1. Similarly, a container-managed relationship between two entity bean classes is treated as a referential integrity constraint between the corresponding entities. The domain constraints specified on a bean translate into the *dom* component of its formula.

For a database schema, with `PRIMARY KEY`, `NOT NULL`, and `UNIQUE` constraints, and `CHECK CONSTRAINT` statements, we generate one formula in our model for each entity (i.e., relation) in the schema. `PRIMARY KEY` statements induce unique and not-null constraints on the appropriate column(s) to be added to the constraint formula. `FOREIGN KEY` statements cause references to be added to the constraint formula and `UNIQUE` constraints are added to formulas of referenced entities.

For domain constraints, we rewrite the formula into disjunctive normal form. Given a constraint involving $\wedge$, $\vee$, and $\neg$, we can apply DeMorgan's Law to revise arithmetic relations to their complements. For example, $(5 < X) \vee \neg((X = 2) \vee (X > 3))$ is converted to $(5 < X) \vee ((X \neq 2) \wedge (X \leq 3))$. The negation of a string constraint, $\mathbf{matchstring}(X, \mathit{expr}, [a, b])$, is defined as all strings that do not satisfy *expr* or those strings that do satisfy *expr* whose length does not fall within the interval $[a, b]$.

4 Ensuring Runtime Integrity

In our architecture, constraint formulas derived from application-level and database specifications are used to generate *guard code* that executes at specific points during EJB-database interactions. The EJB standard provides natural checking points, at each container-database interaction, for the insertion of guard code. Databases offer the option of verifying data integrity at the execution of each `SQL` statement, or of deferring integrity constraint checks until transaction commit time. Programmers may wish to have a similar ability to direct when constraints are checked either by specifying that constraints are verified in immediate or deferred mode, or to have an abstraction that allows control over when and what constraints are checked. We assume that constraints are checked in immediate mode at each container-database interaction. We leave the question of abstractions for supporting programmer-directed checking for future work.

For database integrity constraints, the guard code maintains a runtime data structure called the *transaction shadow database.* The guard code is executed during `EJBLoad`, `EJBStore`, and `EJBRemove` interactions. The EJB standard specifies *callback* methods in every entity bean that are invoked for each of these interactions. We insert invocations to generated guard code in these methods. If the guard code determines that an access is certain to cause an integrity constraint violation in the database, the guard code raises an exception that contains useful information about the source of the error. The application can catch the exception and rectify the error if desired.

Integrity constraints specified at the application-level are checked separately from the database-level integrity constraints whenever a new entity bean instance is created or when the fields of an entity bean instance are updated. In the EJB programming model, each update to a field occurs through the execution of a *set* method (direct accesses to fields are prohibited). Guard code is inserted in these "setter" methods to enforce application integrity constraints. To enforce application-level constraints when an application creates a new entity bean instance, either directly or through an `EJBLoad` operation, we insert guard code in callback methods specified by the EJB standard.

We now describe our architecture for referential integrity and domain constraint checking.

4.1 Referential Integrity Constraints

In the EJB programming model, it is the container's responsibility to ensure that container-managed relationships are used consistently. Application servers currently enforce these application-level integrity constraints by checking that each operation on a container-managed relationship is consistent. They do not, however, monitor referential integrity constraints that have not been manifest as container-managed relationships. We now describe how database referential integrity and uniqueness constraint violations can be detected during EJB-database interactions.

The key data structure underlying our mechanism for application-level verification of referential integrity constraints is the *transaction shadow database.* The transaction shadow database represents the container's (partial) view of the state of the database with respect to a transaction. We shall use the term *shadow database* with the understanding that the lifetime of a shadow database is a transaction. For each relation R of interest in the database, the shadow database contains a shadow table, *shadow(R).* Each shadow table contains entries of the form $\textit{exists}(a_1, \ldots, a_k)$ and $\textit{not-exists}(a_1, \ldots, a_k)$, where the $a_i, 1 \leq i \leq k$, are either integer, float, or string values, the value NULL, or the distinguished value, $*$. We say that a tuple $(a_1, \ldots, a_k)$ *matches* a tuple $(b_1, \ldots, b_k)$ if for all $a_i, 1 \leq i \leq k, (a_i = b_i) \vee (a_i = *)$. Observe that *match* is not a symmetric operator, that is, $match(t, s)$ does not imply $match(s, t)$.

As the container loads and stores data from and to the database within a transaction, it populates the shadow tables with entries. The presence of an entry, $exists(a_1, \ldots, a_k)$, in a shadow database table implies that a tuple matching $(a_1, \ldots, a_k)$ exists in the corresponding relation in the database. Similarly, a tuple $not\text{-}exists(a_1, \ldots, a_k)$ in a shadow database table implies that *no* tuple matching $(a_1, \ldots, a_k)$ exists in the corresponding relation in the database.

The information stored in the shadow database depends on the level of isolation, which may be either *repeatable read* or *serializable* [7]. With an isolation level of *serializable*, one is assured that results of referential integrity checks made on behalf of a transaction's SQL statement remain valid (whether successful or resulting in an error) unless affected by the transaction itself. For example, if a statement fails to insert a tuple t due to a foreign key constraint violation (there is no tuple s corresponding to the key), then a matching tuple will not "spontaneously" appear due to other transaction's actions. With repeatable read, there is no "phantom protection" [4, 7], and therefore, one can only make deductions about data read or updated successfully by the container as such data is locked until the end of the transaction.

Let $t = (t_1, t_2, \ldots, t_n)$ be a tuple over relation schema $R(X_1, \ldots, X_n)$. We define $t' = Proj_Y(t)$, where $Y \subseteq \{X_1, \ldots, X_n\}$ as $(t'_1, t'_2, \ldots, t'_n)$, $t'_i = t_i$ if $X_i \in Y$, and $*$ otherwise (note that *Proj* is *not* the traditional relational projection operator). Let $U \subseteq \{X_1, \ldots, X_n\}$, be a subset of the columns in R declared as UNIQUE. The insertion of a tuple t into the database will definitely violate a uniqueness constraint if there is an entry, $exists(t')$ in the shadow table corresponding to R, such that $match(Proj_U(t), Proj_U(t'))$.

Consider the relations in Figure 1. Assume that the shadow table corresponding to MANAGER consists of two entries, $exists(\text{"Joe"}, \text{"USSales"}, 501, 100000)$ and $not\text{-}exists(*, *, 502, *)$. If the MANAGERID column is marked UNIQUE, then the insertion of the tuple, $(\text{"Sam"}, \text{"USSales"}, 501, 150000)$ into the MANAGER relation will violate the uniqueness integrity constraint.

As another example, consider an entity, $e(\overline{X})$, representing a relation, R. Assume its constraint formula contains a reference, $e'(\overline{Y}) \wedge$ **agree**$((\overline{X'}), (\overline{Y'}))$, where e' represents a relation, R'. The insertion or deletion of a tuple into R will violate a referential integrity constraint if after the insertion or deletion (the precise effects of these actions on the shadow database are detailed later on), there are two entries, $exists(t_1)$, $not\text{-}exists(t_2) \in shadow(R')$, such that $match(Proj_{\overline{Y'}}(t_2), Proj_{\overline{Y'}}(t_1))$. The insertion of a tuple ("Sam", "USSales", 502, 150000, 10000) into the EMPLOYEE relation will violate referential integrity, since it implies the presence of a tuple $(*, \text{"USSales"}, 502, *)$ in $shadow(\texttt{MANAGER})$, which also contains the contradicting tuple $not\text{-}exists(*, *, 502, *)$.

The shadow database is empty at the beginning of a transaction; we describe how it is affected by the execution of each EJB-database interaction. We assume that each entity bean instance, *ejb*, has a reference, $shadow(ejb)$ to the entry for the tuple from which it was populated. Furthermore, each entry, *ent*, in the shadow table of the form $exists(t)$, has a reference $ejb(ent)$ to the entity bean instance that it populated (for simplicity, we assume there is at most one such reference). Under certain circumstances, these references may be `NULL`, as we describe below. We first outline how the shadow database is maintained and used in the absence of *cascading actions*. We then explain how to handle cascades.

EJBLoad(t, R). When a tuple, t, is loaded from a relation R to populate an instance of an entity bean, *ejb*, the container stores an entry, $ent : exists(t)$, in the appropriate shadow table. The tuple loaded may be a projection of the tuple of the relation R in the database. The container stores $*$ for those columns of the tuple for which it does not have values. The EJB instance and the newly-created entry are set to refer to each other by setting the references $shadow(ejb)$ and $ejb(ent)$.

EJBRemove(t, R). Before deleting t from relation R in the database, we check the shadow database to ensure that referential integrity will not be violated. For each relation R' that has a foreign key reference to R, we check $shadow(R')$ to ensure that there is no entry marked *exists* that refers to the tuple being deleted. If there is no such tuple, the delete operation is forwarded to the database.

If the delete operation fails in the database, an appropriate exception is raised. If it succeeds, let T be the subset of columns of R for which the corresponding value in t is not $*$. For each subset of columns U declared unique such that $U \subseteq T$, we remove each entry, $exists(t')$, where $Proj_U(t) = Proj_U(t')$ holds, from the shadow database relation. We also insert $not\text{-}exists(Proj_U(t))$ into the shadow table. Note that no other transaction/application will be able to insert another tuple that matches $Proj_U(t)$. With respect to the current transaction, this tuple will not exist unless it is inserted by the container through an `EJBStore` operation, as will be described.

There is no necessity of checking for referential integrity violations when an application creates an entity bean and deletes it without saving it into the database. We can distinguish this situation by considering the *shadow* reference of the EJB instance. If it is empty, the deletion is of an entity bean whose corresponding tuple is not in the database. Otherwise, the deletion is of an existing database tuple.

EJBStore(t, R). As with the `EJBRemove` case, there are two situations in which an `EJBStore` is performed. In the first case, the application has updated an entity bean that has been populated with values from the database. In the second case, the application creates an entity bean instance that is to be inserted into the database. As mentioned, we can distinguish the two situations by using the *shadow* reference. In both cases, the first two steps are identical. Let T be the subset of columns of R for which the corresponding value in t is not $*$:

1. For each subset of columns U declared unique for R such that $U \subseteq T$, check that there does not exist an entry, $exists(t')$ where $Proj_U(t) = Proj_U(t')$.

2. For each set of columns in R that refers to another relation R', the successful insertion of t into R would imply the presence of a tuple t' (which may contain $*$ values) in $shadow(R')$. Check that the shadow table for R' does not contain *not-exists* entries that would contradict the existence of tuple t' and entail that the insertion of t will fail.

If the `EJBStore` operation updates an existing database tuple, t_{old}, we have to perform an additional step that is similar to performing an `EJBRemove` on a tuple. We must ensure that there is no *exists* entry in a relation that has a foreign key reference to values in columns of t_{old} that are not in t.

If any of these checks fail, we raise an appropriate exception. Otherwise, we proceed with the insertion of the tuple into the database. If the database operation succeeds, we insert the new tuple into the shadow relation, and set the *shadow* and *ejb* references appropriately. We also remove all entries *not-exists(t'')*, where $match(t'', t)$. If the operation is updating an existing tuple, for each relevant subset of columns in R, $U \subseteq T$, that is marked unique and on which t_{old} and t do not agree, we insert $not\text{-}exists(Proj_U(t_{old}))$ into the table.

Deducing Shadow Tuples. We take advantage of the fact that the database is consistent to deduce the existence of tuples in the database. Consider a tuple, t, from a relation, R, that is read from the database, where R has a foreign key reference to relation R'. The following facts about the database will hold until the end of the transaction:

- The current transaction has a lock on t and no other transaction will modify it.
- There exists a tuple, $t' \in R'$, to which t refers. No other transaction will be permitted to delete such a tuple since this would violate database integrity.

For every reference of the form $R'(\overline{Y}) \wedge \textbf{agree}((\overline{X'}), (\overline{Y'}))$ in the constraint formula for $R(\overline{X})$, we can insert an entry $exists(t')$ into the shadow table R', where the columns corresponding to $\overline{Y'}$ in t' obtain their values from the appropriate columns in t, and the remaining columns in t' are set to $*$. Based on R' references, we may insert additional tuples (this resembles steps in a *chase* [15]). There may already be an entry corresponding to this tuple, which can be determined by searching for an entry, $exists(t'')$, where $match(t', t'')$. In this case, we do not add an entry.

A deduced entry does not have an *ejb* reference to an entity bean instance. On an `EJBLoad` of a tuple t from R, if there is an entry $exists(t')$ in $shadow(R)$, where $match(t', t)$ and $ejb(t') =$ `NULL`, we replace $exists(t')$ with $exists(t)$ and set the *ejb* and *shadow* references appropriately.

Similarly, we can deduce facts from the successful completion of an insert or an update in an `EJBStore` operation. The success of the operation reveals the existence of tuples that satisfy referential integrity constraints. The shadow database can be updated to reflect this information. The failure of a store operation also provides clues as to the state of the database. If a specific relation can be identified as the cause of the failure, a *not-exists* entry may be added to reflect this fact. If there is more than one foreign key reference from the inserted tuple and the source of the failure cannot be identified, we only have disjunctive information regarding non-existence, which we ignore for simplicity. We can make similar deductions on the success or failure of an `EJBRemove` operation.

Cascading Actions. Database integrity constraints allow the specification of actions that are executed upon tuple deletion. We must ensure that the effects of these actions are mirrored in the shadow database so that the shadow database remains a faithful view of the database. We discuss how a database deletion is handled in the shadow database; the treatment of update is somewhat more complex and we omit the details.

Suppose a tuple t is deleted from the database. We can either simulate the effects of a cascaded delete before performing the deletion in the database, or propagate the effects of a deletion after a successful deletion in the database. We focus on the latter case, namely propagating the effects of deletions in the database. In propagating the effects of t's deletion, we must handle the following possibilities in the database integrity constraints specification:

- `CASCADE`: We delete all tuples in the shadow database that have a foreign key reference to the tuple being deleted. Their deletion may in turn cascade. Because of cascades, the database deletion may cause a "chain reaction" of deletions of tuples from relations for which no information is maintained in the shadow database. These may in turn cause deletions of tuples for which information *does* exist in the shadow database (possibly rendering such information false). To ensure the accuracy of the shadow database, we must delete all *exists* entries that could conceivably be involved in the chain of deletions. Taking a conservative approach, if the deletion of a tuple t may cascade to relation R, all *exists* entries for relation R are eliminated. This may raise a curious situation, in which we have an EJB e previously loaded by the container but no information concerning it in the shadow database!
- `SET NULL`: The semantics of this operation is to place `NULL` values in the appropriate columns of tuples referring to the tuple begin deleted. We process this operation on the shadow database by (1) Performing a sequence of update operations on the shadow database to the referencing *exists* tuples (that is, setting `NULL`s or $*$ as necessary), and (2) Performing actions described for `EJBRemove` on the deleted tuple.
- `SET DEFAULT`: Handled similarly to set `NULL`.
- `RESTRICT`: If there is a reference to the tuple being deleted in the shadow database, then this deletion will fail at the database and an exception is raised at the application level. So, a successful deletion implies no referencing tuple in the database. We can reflect this fact in the shadow table by adding *not-exists* entries.

Database triggers may also affect tuple deletion. Our treatment of triggers is similar to that of cascaded deletions; we take a conservative approach and invalidate any information that could possibly be affected by triggered actions. As triggers may insert and delete tuples, they may affect negative information of the form *not-exists* as well.

4.2 Domain Constraints Code Generation

The shadow information may not always be sufficient to determine constraint satisfaction; for example, data may be missing. In these cases, we use *approximate checks* that rely on locally available data. Such approximations, while not conclusive, add a degree of assurance that is balanced against the cost of full checks (which are not always possible, for example, in disconnected mode).

Given an entity bean instance of an entity, $e(\overline{X})$, with a constraint formula, ($vars$, $unique$, $vars$, dom), we assume that the formula has been expanded sufficiently to include interactions with the corresponding relation, and other relations of interest. This expansion may be in some sense an approximation since one cannot expand recursive formulas completely. For dom formulas, an *approximation*, $Approx(dom)$, will satisfy, $dom \Rightarrow Approx(dom)$. In other words, unsatisfiability of an approximate constraint guarantees the unsatisfiability of the constraint from which it was derived, but the satisfiability of an approximation provides no such guarantee.

Given a set of clauses, C, observe that $_{c_i \in C} c_i \Rightarrow {}_{c_j \in C'} c_j$, where $C' \subseteq C$. This implies that discarding clauses from a conjunction of clauses results in a valid approximation. A formula in disjunctive normal form, $_{i \leq i \leq n} C_i$ where each C_i is a conjunction of clauses, is approximated by $_{i \leq i \leq n} Approx(C_i)$, where each approximation of a C_i discards some of the clauses in C_i. Given $C = c_1 \wedge \ldots \wedge c_n$, we first discard all clauses that involve type declarations. We now present two approximations for a given C_i, which differ in the precision and runtime cost of the tests:

1. Discard all clauses that use variables from *vars*. At the end of this process, we have a set of clauses that only use variables from $\overline{X}$ and constants. These clauses can be checked using only values from an entity bean instance (a *local test*).
2. Partition the set of clauses C_i into numeric and string constraints. For the numeric constraints, a standard linear/integer constraint solver checks feasibility. String constraints are solved separately.

If we discard all clauses in a conjunction, the formula is trivially satisfied. Given an approximate formula in disjunctive normal form, we can generate code that verifies the satisfiability of each of its clauses. If none of these clauses are satisfiable at runtime, an integrity violation exception is raised.

Numeric Constraints: For an entity, $e(\overline{X})$, given a conjunction of numeric constraints, we can derive maximum and minimum values for each of the variables using linear/integer programming. We will generally adopt a simpler approach — when a variable participates only in predicates involving relational operators, it is straightforward to derive maximum and minimum values for that variable. Each disjunct in the predicate represents either a collection of intervals, a point, or the empty set. For example, $(5 < X)$ represents the interval of values less than 5 (extending to $-\infty$), and $((X \neq 2) \wedge (X \leq 3))$ represents the intervals $(X < 2)$, $(X > 2 \wedge X \leq 3)$. The minimum and maximum values can be used to detect illegal values efficiently.

If a numeric constraint formula only involves variables from $\overline{X}$, we can generate code that evaluates these constraints using values from the entity bean instance at runtime. If the constraint contains other variables, a constraint solver can be used to ensure feasibility. If an entity, e, has a reference to an entity, e', and the constraint formula for e refers to variables from e', we can generate code that checks at runtime:

1. If, for an instance of e, the shadow database contains values for the corresponding instance of e', the constraint is checked using values from the shadow database.
2. Otherwise, the generated code uses the approximation techniques discussed earlier.

String Constraints: We can generate code that verifies that a string belongs to the language of the expression associated with a constraint. The string length can be checked against the bounds on the constraint to detect violations quickly.

Example 4. Consider the `EMPLOYEE` relation in Figure 1, and an entity bean instance `emp` with persistent fields {`name`, `dept`, `mgrid`, `salary`} that correspond to columns in `EMPLOYEE`. We expand the formula for the entity, $emp(\overline{X})$, to include its interactions with the `EMPLOYEE` relation. After simplification by equality propagation, the expanded constraint formula is (we omit the type checking components of the *dom* formulas for space):

vars : $\{\overline{Y}, \overline{U}\}$
unique : $\{\mathbf{unique}(\texttt{MANAGER}, \overline{U}, \{U_2, U_3\})\}$
refs : $\{\texttt{EMPLOYEE}(\overline{Y}) \wedge \mathbf{agree}(\overline{X}, \overline{Y'})\}$
dom : $(X_1 \neq \texttt{NULL}) \wedge (X_2 = \text{“US*”}) \wedge (Y_5 \leq 0.4 \times X_4) \wedge (X_2 \neq \text{“USSales”} \vee (X_4 \leq U_4))$

where $\overline{X} = \{X_1, \ldots, X_4\}, \overline{Y} = \{Y_1, \ldots, Y_5\}, \overline{Y'} = \{Y_1, \ldots, Y_4\}, \overline{U} = \{U_1, \ldots, U_4\}$, and the **agree** clause equates X_i to $Y_i, 1 \leq i \leq 4$.

In the formula, $\overline{Y}$ represents the columns of the `EMPLOYEE` relation, and $\overline{U}$ represents the columns of the `MANAGER` relation to which the `EMPLOYEE` relation refers. We can approximate *dom* by removing clauses involving variables not local to `emp`, that is, the clauses involving Y_5 and U_4. For each remaining clause, we generate code in a straightforward fashion to obtain:

```
boolean checkEmp (String n, String d, int m, int s) {
 if (n = NULL)
  return false // C1: Name must be non-NULL
 if (d[0] ≠ 'U' || d[1] ≠ 'S')
  return false // C2: Dept must start with ``US''
 return true
}
```

As mentioned earlier, we plan to use static analysis in the future to simplify (or eliminate) the generated checks. For example, if static analysis can establish that $n \neq \texttt{NULL}$, then the above check for $(n = \texttt{NULL})$ will not be generated.

Example 5. Continuing the previous example, we can obtain greater precision in our checks by a deeper expansion. If the `EMPLOYEE` entity were expanded first with respect to the `MANAGER` entity, and then, the `emp` entity were expanded with respect to the expanded `EMPLOYEE` formula, we obtain for the *dom* component (after simplification):

$dom : (X_1 \neq \texttt{NULL}) \wedge (X_2 = \text{``US*''}) \wedge (Y_5 \leq 0.4 \times X_4) \wedge (X_2 \neq \text{``USSales''} \vee (X_4 \leq U_4)) \wedge (U_1 \neq \texttt{NULL}) \wedge (500 \leq X_3) \wedge (X_3 \leq 999) \wedge (100000 \leq U_4) \wedge (U_4 \leq 200000)$

We can now add the following lines to `checkEmp` to increase the precision of our checks:

- `if (`$m < 500 \,\|\, m > 999$`) return false;` This condition is derived from the constraint C8 in Figure 1.
- `if (`d` = ''USSales'' && `$X_4 > 200000$`) return false;` This condition checks the feasability of $X_4 \leq U_4$ when $X_2 =$"USSales", given $100000 \leq U_4 \leq 200000$ (C9 in Figure 1).

Even without access to values of the `MANAGER` relation referred to by the `EMPLOYEE` relation corresponding to `emp`, we can determine the feasibility of constraints on the `MANAGER` relation. In general, a constraint solver may be necessary at runtime to determine the feasibility of the set of constraints. Depending on the level of precision desired, we can choose to generate code to invoke a constraint solver at runtime. If a shadow tuple corresponding to the appropriate `MANAGER` (represented by the set of variables $\overline{U}$ in the formula) is available, we can use this information to check that if the `emp`'s department is "USSales," the salary of `emp` is less than the manager's salary.

5 Conclusions

At application development time, current programming frameworks offer little support for specifying application data constraints and integrating these constraints with those specified on other applications or a database. A programmer must insert explicit checks to enforce constraints, which is an error-prone approach. Aside from the complexity of programming these constraints, the cost of runtime integrity violations may be high, especially when checking is deferred until transaction commit time.

We have presented an architecture that represents integrity constraints derived from specifications of database and application integrity constraints, and mappings between database relations and application objects, in a common formalism. From the constraints in the formalism, code is generated automatically to verify integrity contstraints at runtime. The verification of constraints relies on a *shadow database*, a narrow window to the state of the database, to detect, at runtime, database accesses that are guaranteed to cause an integrity violation. We also discuss how constraint formulas may be expanded and approximated to enable the verification of domain integrity constraints statically and dynamically. Even when only partial data is available, these techniques can detect many violations.

For future work in the DOMO (Data-Object Modeling and Optimization) project at IBM Research, we plan to extend our approach to other forms of application-database interactions, especially for data sources based on XML schemas with integrity constraints. We also plan to leverage static analysis for enhanced checking of integrity constraints.

Acknowledgments

We thank the anonymous reviewers for their valuable comments.

References

1. U. Banerjee. *Dependence analysis for supercomputing.* Kluwer Academic, Boston, MA, USA, 1988.
2. P. A. Bernstein and B. T. Blaustein. Fast methods for testing quantified relational calculus assertions. In *Proceedings of SIGMOD*, pages 39–50, June 1982.
3. P. A. Bernstein, B. T. Blaustein, and E. M. Clarke. Fast maintenance of semantic integrity assertions using redundant aggregate data. In *Proceedings of VLDB*, pages 126–136, October 1980.
4. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
5. R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *Proceedings of PLDI*, pages 321–333, 2000.
6. A. Brucker and B. Wolff. Checking OCL constraints in distributed component based systems. Technical Report 157, Institut fur Informatik, Universitat Freiburg, 2001.
7. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.
8. P. Grefen and J. Widom. Integrity constraint checking in federated databases. In *Proceedings of the Intl. Conf. on Cooperative Information Systems*, 1996.
9. A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom. Constraint checking with partial information. In *Proceedings of PODS*, pages 45–55, 1994.
10. A. Gupta and J. Widom. Local verification of global integrity constraints. In *Proceedings of SIGMOD*, pages 49–58, June 1993.
11. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
12. A. N. S. Institute. Information technology—database languages—SQL. ANSI X3.135-1992, 1992.
13. H. V. Jagadish and X. Qian. Integrity maintenance in an object-oriented database. In *Proceedings of VLDB*, pages 469–480, 1982.
14. K. E. Kline and D. L. Kline. *SQL in a Nutshell.* O'Reilly and Associates, Inc., 2001.
15. D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.
16. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.
17. R. M. Roos. *Java Data Objects.* Addison-Wesley, 2002.
18. D. Sceppa. *The Microsoft ADO.NET (Core Reference).* Microsoft Press, 2002.

19. Sun Microsystems, Inc. *Enterprise Java Beans Technology.* http://www.javasoft.com/j2ee.
20. J. Warmer and A. Kleppe. *The Object Constraint Language, Precise Modeling with UML.* Addison-Wesley, 1999.

A Constraint Model Semantics

The predicate $\mathbf{agree}((X_1,\ldots,X_k),(Y_1,\ldots,Y_k))$ means equalities on corresponding elements of the sequences, that is, $X_1 = Y_1 \wedge \ldots \wedge X_k = Y_k$. A constraint formula, C, in our formalism can be viewed as a first-order logic sentence:

$$\begin{aligned}
&\forall\overline{X}[e(\overline{X}) \Rightarrow \exists\overline{Y_1},\ldots,\overline{Y_k},\ \mathbf{unique}(ent,\overline{Z},\overline{Z'}) \wedge \ldots\\
&\qquad\qquad \wedge e_i(\overline{Y_i})\wedge\ \mathbf{agree}((\overline{X_i'}),(\overline{Y_i'})) \wedge \ldots\\
&\qquad\qquad \wedge\ dom(\overline{X},\overline{Y_1},\ldots,\overline{Y_k})]
\end{aligned}$$

where $\{\overline{Y_1},\ldots,\overline{Y_i},\ldots,\overline{Y_k}\}\ = vars$,
and $\overline{Z'} \subseteq \overline{Z}$, where $\overline{Z} \in vars\ \cup\ \{\overline{X}\}$.

The constraint formula, C, is satisfied if for all assignments of values to the variables in $\overline{X}$, the existence of a tuple t in a relation associated with an entity, e, with the values assigned to $\overline{X}$, implies that there is an assignment of values to each $\overline{Y} \in vars$ such that the following holds:

1. Each clause of the form, $\mathbf{unique}(ent,\overline{Z},\overline{Z'}),\overline{Z'} \subseteq \overline{Z}$, means that there can be no two distinct tuples in *ent* that have the same values for the columns in $\overline{Z'}$. (In fact, columns $\overline{Z'}$ form a key in entity *ent*.)
2. Each element of the form, $e'(\overline{Y})\wedge$ $\mathbf{agree}((\overline{X'}),(\overline{Y'}))$ means that there is a tuple in the relation associated with entity e' with the values assigned to $\overline{Y}$; furthermore, the sequence of values assigned to $\overline{X'} \subseteq \overline{X}$ and the sequence of values assigned to $\overline{Y'} \subseteq \overline{Y}$, agree pairwise.
3. Predicate *dom* is satisfied using values assigned to $\overline{X},\overline{Y_1},\ldots,\overline{Y_k}$.

A Unifying Semantics for Active Databases Using Non-Markovian Theories of Actions

Iluju Kiringa[1] and Ray Reiter[2]

[1] School of Information Technology and Engineering, University of Ottawa
kiringa@site.uottawa.ca
http://www.site.uottawa.ca/~kiringa
[2] Department of Computer Science, University of Toronto
http://www.cs.toronto.edu/DCS/People/Faculty/reiter.html

Abstract. Over the last fifteen years, database management systems (DBMSs) have been enhanced by the addition of rule-based programming to obtain active DBMSs. One of the greatest challenges in this area is to formally account for all the aspects of active behavior using a uniform formalism. In this paper, we formalize active relational databases within the framework of the situation calculus by uniformly accounting for them using theories embodying non-Markovian control in the situation calculus. We call these theories *active relational theories* and use them to capture the dynamics of active databases. Transaction processing and rule execution is modelled as a theorem proving task using active relational theories as background axioms. We show that major components of an ADBMS may be given a clear semantics using active relational theories.

1 Introduction

Over a large portion of the last thirty years, database management systems (DBMSs) have generally been passive in the sense that only users can perform definition and manipulation operations on stored data. In the last fifteen years, they have been enhanced by the addition of active behavior to obtain active DBMSs (ADBMSs). Here, the system itself performs some definition and manipulation operations automatically, based on a suitable representation of the (re)active behavior of the application domain and the operations performed during a database transaction.

The concept of rule and its execution are essential to ADBMSs. An ADBMS has two major components, a *representational* component called the rule language, and an *executional* component called the execution model. The rule language is used to specify the active behavior of an application. Typical rules used here are the so-called EVENT-CONDITION-ACTION (ECA) rules which are a syntactical construct representing the notion that an action must be performed upon detection of a specified event, provided that some specified condition holds. The execution model is intimately related to the notion of database transaction. A database transaction is a (sometimes nested or detached) sequence of database update operations such as *insert*, *delete*, and *update*, used to insert tuples into, delete them from, or update them in a database. The execution model comes in three flavors: immediate execution, deferred execution, and detached

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 110–129, 2004.

execution, meaning that rules are executed interleaved with database update operations, at the end of user transactions, and in a separate transaction, respectively.

It has been recognized in the literature that giving a formal foundation to active databases is a hard challenge [31, 10, 7, 2]. Different formalisms have been proposed for modelling parts of the concept of rule and its execution (See, e.g., [10, 7, 24]). Very often however, different parts have been formalized using different formalisms, which makes it very difficult to globally reason about active databases.

In this paper, we give a formalization of the concept of active database within the framework of the situation calculus. Building on [18] and [27], this paper aims to account formally for active databases as a further extension of the relational databases formalized as relational theories to accommodate new world knowledge, in this case representational issues associated with rules, the execution semantics, and database transactions.

Including dynamic aspects into database formalization has emerged as an active area of research. In particular, a variety of logic-based formalization attempts have been made [30, 13, 5, 27, 4]. Among these, we find model theoretic approaches (e.g., [30]), as opposed to syntactic approaches (e.g., [13, 27]).

Proposals in [27], and [4] use the language of the situation calculus [21, 28]. This language constitutes a framework for reasoning about actions that relies on an important assumption: the execution preconditions of primitive actions and their effects depend solely on the current situation. This assumption is what the control theoreticians call the Markov property. Thus non-Markovian actions are precluded in the situation calculus used in those proposals. However, in formalizing database transactions, one quickly encounters settings where using non-Markovian actions and fluents is unavoidable. For example, a transaction model may explicitly execute a $Rollback(s)$ to go back to a specific database state s in the past; a *Commit* action is executable only if the previous database states satisfy a set of given integrity constraints and there is no other committing state between the beginning of the transaction and the current state; and an event in an active database is said to have occurred in the current database state if, in some database state in the past, that event had occurred and no action changed its effect meanwhile. Thus one clearly needs to address the issue of non-Markovian actions and fluents explicitly when formalizing database transactions, and researchers using the situation calculus or similar languages to account for updates and active rules fail to do that. Our framework will therefore use non-Markovian control [11].

Though several other similar logics (e.g. dynamic logic, event calculus, and transaction logic) could have been used for our purpose, we prefer the situation calculus, due to a set of features it offers, the most beneficial of which are: the treatment of actions as first class citizens of the logic, thus allowing us to remain within the language to reason about actions; the explicit addressing of the frame problem that inevitably occurs in the context of the database updates; and perhaps most important of all, the relational database log is a first class citizen in the logic. Without a treatment of logs as first class citizens, there seems to be no obvious way of expressing properties about them in the form of logical sentences.

The main contributions of the formalization reported in this paper can succinctly be summarized as follows:

1. We construct logical theories called active relational theories to formalize active databases along the lines set by the framework in [15]; active relational theories are non-Markovian theories in which one may explicitly refer to all past states, and not only to the previous one. They provide the formal semantics of the corresponding active database model. They are an extension of the classical relational theories of [26] to the transaction and active database settings.

2. We give a logical tool that can be used to classify various execution models, and demonstrate its usefulness by classifying execution models of active rules that are executed in the context of flat database transactions.

The remainder of this paper is organized as follows. Section 2 introduces the situation calculus. Section 3 formalizes database transactions as non-Markovian theories of the situation calculus. In Section 4, both the ECA rule paradigm and the main execution models of active rules will be given a logical semantics based on an extension of theories introduced in Section 3. In Section 5, we classify the execution models. Finally, Section 6 briefly reviews related work, concludes the paper and mentions future work.

2 The Situation Calculus

The situation calculus [22], [28] is a many-sorted second order language for axiomatizing dynamic worlds. Its basic ingredients consist of *actions*, *situations* and *fluents*; its universe of discourse is partitioned into sorts for actions, situations, and objects other than actions and situations.

Actions are first order terms consisting of an action function symbol and its arguments. In modelling databases, these correspond to the elementary operations of inserting, deleting and updating relational tuples. For example, in the stock database (Example 1, adapted from [29]) that we shall use below, we have the first order term $price\_insert(stock\_id, price, time, trans)$ that denotes the operation of inserting the tuple $(stock\_id, price, time)$ into the database relation $price$ by the transaction $trans$.

A situation is a first order term denoting a sequence of actions. These sequences are represented using a binary function symbol do: $do(\alpha, s)$ denotes the sequence resulting from adding the action α to the sequence s. So $do(\alpha, s)$ is like LISP's $cons(\alpha, s)$, or Prolog's $[\alpha \mid s]$. The special constant S_0 denotes the *initial situation*, namely the empty action sequence, so S_0 is like LISP's ( ) or Prolog's [ ]. In modelling databases, situations will correspond to the database *log*[1].

Relations and functions whose values vary from state to state are called *fluents*, and are denoted by predicate or function symbols with last argument a situation term. In Example 1 below, $price(stock\_id, price, time, trans, s)$ is a relational fluent, meaning that in that database state that would be reached by performing the sequence of operations in the log s, $(stock\_id, price, time)$ is a tuple in the $price$ relation, inserted there by the transaction $trans$.

[1] In fact, in the database setting, a situation is a sequence involving many histories corresponding to as many transactions.

Example 1. Consider a stock database (adapted from [29]) whose schema has the relations:

$$\begin{aligned}&price(stock\_id, price, time, trans, s),\\ &stock(stock\_id, price, closingprice, trans, s), \text{ and}\\ &customer(cust\_id, balance, stock\_id, trans, s),\end{aligned}$$

which are relational fluents. The explanation of the attributes is as follows: $stock\_id$ is the identification number of a stock, $price$ the current price of a stock, $time$ the pricing time, $closingprice$ the closing price of the previous day, $cust\_id$ the identification number of a customer, $balance$ the balance of a customer, and $trans$ is a transaction identifier. □

Notice that, contrary to the presentation in [28], a fluent now contains a further argument – which officially is its second last argument – specifying which transaction contributed to its truth value in a given log. The domain of this new argument could be arbitrarily set to, e.g., integers.

In addition to the function *do*, the language also includes special predicates $Poss$, and $\sqsubset$. $Poss(a(\boldsymbol{x}), s)$ means that the action $a(\boldsymbol{x})$ is possible in the situation s; and $s \sqsubset s'$ states that the situation s is a prefixing sublog of situation s'. By stating that $s \sqsubset s'$, nothing is said about the possibility of actions that constitute s and s'. For instance,

$$Poss(price\_delete(ST1, \$100, 4PM, 1), S_0)$$

and

$$S_0 \sqsubset do(price\_delete(ST1, \$100, 4PM, 2), S_0)$$

are ground atoms of $Poss$ and $\sqsubset$, respectively.

The set $\mathfrak{W}$ of well formed formulas (wffs) of the situation calculus, together with terms, atomic formulas, and sentences are defined in the standard way of second order languages. Additional logical constants are introduced in the usual way.

In [27], it is shown how to formalize a dynamic relational database setting in the situation calculus with axioms that capture change which are: action precondition axioms stating when database updates are possible, successor state axioms stating how change occurs, unique name axioms that state the uniqueness of update names, and axioms describing the initial situation. These axioms constitute a *basic action theory*, in which control over the effect of the actions in the next situation depends solely on the current situation. This was achieved by precluding the use of the predicate $\sqsubset$ in the axioms. We extend these theories to capture active databases by incorporating non-Markovian control. We achieve this by using the predicate $\sqsubset$ in the axioms.

For simplicity, we consider only primitive update operations corresponding to insertion or deletion of tuples into relations. For each such relation $F(\boldsymbol{x}, t, s)$, where $\boldsymbol{x}$ is a tuple of objects, t is a transaction argument, and s is a situation argument, a *primitive internal action* is a parameterized primitive action of the situation calculus of the form $F\_insert(\boldsymbol{x}, t)$ or $F\_delete(\boldsymbol{x}, t)$.

We distinguish the primitive internal actions from *primitive external actions*, namely $Begin(t)$, $Commit(t)$, $End(t)$, and $Rollback(t)$, whose meaning will be clear in the sequel of this paper; these are external as they do not specifically affect the content of the database[2]. The argument t is a unique transaction identifier. Finally, the set of fluents is partitioned into two disjoint sets, namely a set of *database fluents* and a set of *system fluents*. Intuitively, the database fluents represent the relations of the database domain, while the system fluents are used to formalize the processing of the domain. Usually, any functional fluent in a relational language will always be a system fluent.

Now, in order to represent relational databases, we need some appropriate restrictions on the situation calculus.

Definition 1. *A* basic relational language *is a subset of the situation calculus whose alphabet* $\mathfrak{A}$ *is restricted to (1) a finite number of constants, but at least one, (2) a finite number of action functions, (3) a finite number of functional fluents, and (4) a finite number of relational fluents.*

3 Specifying Database Transactions

This section introduces a characterization of flat transactions in terms of theories of the situation calculus. These theories give axioms of flat transaction models that constrain database logs in such a way that these logs satisfy important correctness properties of database transaction, including the so-called ACID properties[12].

Definition 2. (Flat Transaction) *A sequence of database actions is a* flat transaction *iff it is of the form* $[a_1, \ldots, a_n]$*, where the* a_1 *must be* $Begin(t)$*, and* a_n *must be either* $Commit(t)$*, or* $Rollback(t)$*;* $a_i, i = 2, \cdots, n-1$*, may be any of the primitive actions, except* $Begin(t)$*,* $Rollback(t)$*, and* $Commit(t)$*; here, the argument* t *is a unique identifier for the atomic transaction.*

Notice that we do not introduce a term of a new sort for transactions, as is the case in [4]; we treat transactions as run-time activities — execution traces — whose design-time counterparts will be ConGOLOG programs introduced in the next chapter. We refer to transactions by their names that are of sort *object*. Notice also that, on this definition, a transaction is a semantical construct which will be denotations of situations of a special kind called legal logs in the next section.

The axiomatization of a dynamic relational database with flat transaction properties comprises the following classes of axioms:

Foundational Axioms. These are constraints imposed on the structure of database logs [25]. They characterize database logs as finite sequences of updates and can be proved to be valid sentences.

Integrity Constraints. These are constraints imposed on the data in the database at a given situation s; their set is denoted by $\mathcal{IC}_e$ for constraints that must be enforced at each update execution, and by $\mathcal{IC}_v$ for those that must be verified at the end of the flat transaction.

[2] The terminology internal versus *external* action is also used in [20], though with a different meaning.

Update Precondition Axioms. There is one for each internal action $A(\boldsymbol{x}, t)$, with syntactic form

$$Poss(A(\boldsymbol{x}, t), s) \equiv (\exists t')\mathit{\Pi}_A(\boldsymbol{x}, t', s) \wedge IC_e(do(A(\boldsymbol{x}, t), s)) \wedge running(t, s). \quad (1)$$

Here, $\mathit{\Pi}_A(\boldsymbol{x}, t, s)$ is a first order formula with free variables among $\boldsymbol{x}, t$, and s. Moreover, the formula on the right hand side of (1) is uniform in s[3]. Such an axiom characterizes the precondition of the update A. Intuitively, the conjunct $\mathit{\Pi}_A(\boldsymbol{x}, t', s)$ is the part of the right hand side of (1) that really gives the precondition for executing the update $A(\boldsymbol{x}, t)$, $IC_e(do(A(\boldsymbol{x}, t), s))$ says that the integrity constraints $\mathcal{IC}_e$ are enforced in the situation resulting from the execution of $A(\boldsymbol{x}, t)$ in s, and $running(t, s)$ says that the transaction t has not terminated yet. Formally, $IC_e(s)$ and $running(t, s)$ are defined as follows:

$$\begin{aligned} &IC_e(s) =_{df} \mathop{}\limits_{IC \in \mathcal{IC}_e} IC(s). \\ &running(t, s) =_{df} (\exists s').do(Begin(t), s') \sqsubseteq s \wedge \\ &\qquad (\forall a, s'')[do(Begin(t), s') \sqsubset do(a, s'') \sqsubset s \supset a \neq Rollback(t) \wedge a \neq \\ &\qquad\qquad End(t)]. \end{aligned}$$

In the stock example, the following states the condition for deleting a tuple from the *customer* relation:

$$\begin{aligned} &Poss(customer\_delete(cid, bal, sid, t), s) \equiv \\ &\qquad (\exists t')customer(cid, bal, sid, t', s) \wedge \\ &\qquad IC_e(do(customer\_delete(cid, bal, sid, t), s)) \wedge running(t, s). \end{aligned} \quad (2)$$

Successor State Axioms. These have the syntactic form

$$\begin{aligned} F(\boldsymbol{x}, t, do(a, s)) \equiv & (\exists \boldsymbol{t_1})\mathit{\Phi}_F(\boldsymbol{x}, a, \boldsymbol{t_1}, s) \wedge \neg(\exists t'')a = Rollback(t'') \vee \\ & (\exists t'')a = Rollback(t'') \wedge restoreBeginPoint(F, \boldsymbol{x}, t'', s), \end{aligned} \quad (3)$$

There is one such axiom for each database relational fluent F. The formula on the right hand side of (3) is uniform in s, and $\mathit{\Phi}_F(\boldsymbol{x}, a, \boldsymbol{t}, s)$ is a formula with free variables among $\boldsymbol{x}, a, \boldsymbol{t}, s$; $\mathit{\Phi}_F(\boldsymbol{x}, a, \boldsymbol{t}, s)$ specifies how changes occur with respect to internal actions and has the canonical form

$$\gamma^{+}_{F}(\boldsymbol{x}, a, \boldsymbol{t}, s) \vee F(\boldsymbol{x}, s) \wedge \neg\gamma^{-}_{F}(\boldsymbol{x}, a, \boldsymbol{t}, s), \quad (4)$$

where $\gamma^{+}_{F}(\boldsymbol{x}, a, \boldsymbol{t}, s)$ ($\gamma^{-}_{F}(\boldsymbol{x}, a, \boldsymbol{t}, s)$) denotes a first order formula specifying the conditions that make a fluent F true (false) in the situation following the execution of a [28].

Formally, $restoreBeginPoint(F, \boldsymbol{x}, t, s)$ is defined as follows:

[3] A formula $\phi(s)$ is uniform in a situation term s if s is the only situation term that all the fluents occurring in $\phi(s)$ mention as their last argument.

Abbreviation 1

$$
\begin{array}{l}
\mathit{restoreBeginPoint}(F,\boldsymbol{x},t,s) =_{\mathit{df}}\\
\quad\{(\exists a_1,a_2,s',s_1,s_2,t').\\
\quad\quad do(Begin(t),s') \sqsubset do(a_2,s_2) \sqsubset do(a_1,s_1) \sqsubseteq s \wedge\\
\quad\quad\quad\quad writes(a_1,F,\boldsymbol{x},t) \wedge writes(a_2,F,\boldsymbol{x},t') \wedge\\
\quad\quad [(\forall a'',s'').do(a_2,s_2) \sqsubset do(a'',s'') \sqsubset do(a_1,s_1) \supset \neg writes(a'',F,\boldsymbol{x},t)] \wedge\\
\quad\quad [(\forall a'',s'').do(a_1,s_1) \sqsubseteq do(a'',s'') \sqsubset s \supset\\
\quad\quad\quad\quad \neg(\exists t'')writes(a'',F,\boldsymbol{x},t'')] \wedge (\exists t'')F(\boldsymbol{x},t'',s_1)]\} \vee\\
\quad\{(\forall a^*,s^*,s').do(Begin(t),s') \sqsubset do(a^*,s^*) \sqsubseteq s \supset\\
\quad\quad\quad\quad\quad \neg writes(a^*,F,\boldsymbol{x},t)] \wedge (\exists t')F(\boldsymbol{x},t',s)\}.
\end{array}
$$

Notice that system fluents have successor state axioms that do not necessarily have the form (3). Intuitively, $\mathit{restoreBeginPoint}(F,\boldsymbol{x},t,s)$ means that the system restores the value of the database fluent F with arguments $\boldsymbol{x}$ in a particular way that captures the semantics of $\mathit{Rollback}$:

- The first disjunct in Abbreviation 1 captures the scenario where the transactions t and t' running in parallel, and writing into and reading from F are such that t overwrites whatever t' writes before it (t) rolls back. Suppose that t and t' are such that t begins, and eventually writes into F before rolling back; t' begins after t has begun, writes into F before the last write action of t, and commits before t rolls back. Now the second disjunct in 1 says that the value of F must be set to the "before image" [3] of the first $w(t)$, that is, the value the F had just before the first $w(t)$ was executed.
- The second disjunct in Abbreviation 1 captures the case where the value F had at the beginning of the transaction that rolls back is kept.

Given the actual situation s, the successor state axiom characterizes the truth values of the fluent F in the next situation $do(a,s)$ in terms of all the past situations. Notice that Abbreviation 1 captures the intuition that $\mathit{Rollback}(t)$ affects all tuples within a table. As an example, the following gives a successor state axiom for the fluent $\mathit{customer}(\mathit{cid},\mathit{bal},\mathit{stid},\mathit{tr},s)$.

$$
\begin{array}{l}
\mathit{customer}(\mathit{cid},\mathit{bal},\mathit{stid},t,do(a,s))\\
\quad\equiv ((\exists t_1)a = \mathit{customer\_insert}(\mathit{cid},\mathit{bal},\mathit{stid},t_1) \vee\\
\quad\quad (\exists t_2)\mathit{customer}(\mathit{cid},\mathit{bal},\mathit{stid},t_2,s) \wedge\\
\quad\quad \neg(\exists t_3)a = \mathit{customer\_delete}(\mathit{cid},\mathit{bal},\mathit{stid},t_3)) \wedge \neg(\exists t')a = \mathit{Rollback}(t') \vee\\
\quad (\exists t').a = \mathit{Rollback}(t') \wedge \mathit{restoreBeginPoint}(\mathit{customer},(\mathit{cid},\mathit{bal},\mathit{stid}),t',s).
\end{array}
$$

In this successor state axiom, the formula

$$
\begin{array}{l}
(\exists t_1)a = \mathit{customer\_insert}(\mathit{cid},\mathit{bal},\mathit{stid},t_1) \vee\\
\quad (\exists t_2)\mathit{customer}(\mathit{cid},\mathit{bal},\mathit{stid},t_2,s) \wedge \neg(\exists t_3)a = \mathit{customer\_delete}(\mathit{cid},\mathit{bal},\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{stid},t_3)
\end{array}
$$

corresponds to the canonical form (4).

Precondition Axioms for External Actions. This is a set of action precondition axioms for the transaction specific actions $Begin(t)$, $End(t)$, $Commit(t)$, and $Rollback(t)$. The external actions of flat transactions have the following precondition axioms[4]:

$$Poss(Begin(t), s) \equiv \neg(\exists s')do(Begin(t), s') \sqsubseteq s, \tag{5}$$

$$Poss(End(t), s) \equiv running(t, s), \tag{6}$$

$$\begin{aligned} &Poss(Commit(t), s) \equiv (\exists s').s = do(End(t), s')\wedge \\ &\underset{IC\in\mathcal{IC}_v}{}\quad IC(s) \wedge (\forall t')[sc\_dep(t, t', s) \supset (\exists s'')do(Commit(t'), s'') \sqsubseteq s], \end{aligned} \tag{7}$$

$$\begin{aligned} &Poss(Rollback(t), s) \equiv (\exists s')[s = do(End(t), s')\wedge \\ &\neg \underset{IC\in\mathcal{IC}_v}{}\quad IC(s)] \vee (\exists t', s'')[r\_dep(t, t', s) \wedge do(Rollback(t'), s'') \sqsubseteq s]. \end{aligned} \tag{8}$$

Notice that our axioms (5)–(8) assume that the user will only use internal actions $Begin(t)$ and $End(t)$ and the system will execute either $Commit(t)$ or $Rollback(t)$. Intuitively, the predicates $sc\_dep(t, t', s)$ and $r\_dep(t, t', s)$ means that transaction t is is strong commit dependent on transaction t', and that transaction t is is rollback dependent on transaction t', respectively[5].

Dependency axioms. These are the following transaction model-dependent axioms:

$$r\_dep(t, t', s) \equiv transConflict(t, t', s), \tag{9}$$

$$sc\_dep(t, t', s) \equiv readsFrom(t, t', s). \tag{10}$$

The defined predicates $r\_dep(t, t', s)$, $sc\_dep(t, t', s)$ are called dependency predicates. The first axiom says that a transaction conflicting with another transaction generates a rollback dependency, and the second says that a transaction reading from another transaction generates a strong commit dependency.

Unique Names Axioms. These state that the primitive updates and the objects of the domain are pairwise unequal.

Initial Database. This is a set of first order sentences specifying the initial database state. They are completion axioms of the form

$$(\forall \boldsymbol{x}, t).F(\boldsymbol{x}, t, S_0) \equiv \boldsymbol{x} = \boldsymbol{C}^{(1)} \vee \ldots \vee \boldsymbol{x} = \boldsymbol{C}^{(r)}, \tag{11}$$

one for each (database or system) fluent F. Here, the $\boldsymbol{C}^i$ are tuples of constants. Also, $\mathcal{D}_{S_0}$ includes unique name axioms for constants of the database. Axioms of the form

[4] It must be noted that, in reality, a part of rolling back and committing lies with the user and another part lies with the system. So, we could in fact have something like $Rollback_{sys}(t)$ and $Commit_{sys}(t)$ on the one hand, and $Rollback_{usr}(t)$ and $Commit_{usr}(t)$ on the other hand. However, the discussion is simplified by considering only the system's role in executing these actions.

[5] A transaction t is rollback depend on transaction t' iff, whenever t' rolls back in a log, then t must also roll back in that log; t is strong commit depend on t' iff, whenever t' commits in s, then t must also commit in s.

(11) say that our theories accommodate a complete initial database state, which is commonly the case in relational databases as unveiled in [26]. This requirement is made to keep the theory simple and to reflect the standard practise in databases. It has the theoretical advantage of simplifying the establishment of logical entailments in the initial database; moreover, it has the practical advantage of facilitating rapid prototyping of the ATMs using Prolog which embodies negation by failure, a notion close to the completion axioms used here.

Definition 3. (Basic Relational Theory) *Suppose* $\mathfrak{R} = (\mathfrak{A}, \mathfrak{W})$ *is a basic relational language. Then a theory* $\mathcal{D} \subseteq \mathfrak{W}$ *that comprises the axioms of the form described above is a* basic relational theory.

4 Formalizing Active Databases

4.1 GOLOG

GOLOG [18] is a situation calculus-based programming language for defining complex actions in terms of a set of primitive actions axiomatized in the situation calculus according to Section 2. It has the standard – and some not so standard – control structures found in most Algol-like languages: *Sequence* ($[\alpha\,;\,\beta]$; Do action α, followed by action β); *Test actions* ($p?$; Test the truth value of expression p in the current situation); *Nondeterministic action choice* ($\alpha \mid \beta$; Do α or β); *Nondeterministic choice of arguments* ($(\pi\ x)\alpha$; nondeterministically pick a value for x, and for that value of x, do action α); *Conditionals* and *while* loops; and *Procedures*, including recursion.

In [8], GOLOG has been enhanced with parallelism to obtain ConGOLOG.

With the ultimate goal of capturing active databases as theories extending basic relational theories, it is appropriate to adopt an operational semantics of GOLOG programs based on a single-step execution of these programs; such a semantics is introduced in [8]. First, two special predicates $Trans$ and $Final$ are introduced. $Trans(\delta, s, \delta', s')$ means that program δ may perform one step in situation s, ending up in situation s', where program δ' remains to be executed. $Final(\delta, s)$ means that program δ may terminate in situation s. A single step here is either a primitive or a testing action. Then the two predicates are characterized by appropriate axioms.

Program execution is captured by using the abbreviation $Do(\delta, s, s')$ [28]. Single-stepping a given program δ means executing the first primitive action of δ, leaving a remaining fragment δ' of δ to be executed. $Do(\delta, s, s')$ intuitively means that, beginning in a given situation s, program δ is single-stepped until the ultimate remainder of program δ may terminate in situation s'.

Formally, we have [8]:

$$Do(\delta, s, s') =_{df} (\exists \delta').Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s'),$$

where $Trans^*$ denotes the transitive closure of $Trans$.

In the sequel, we shall develop a syntax for capturing the concept of ECA rule in our framework. After that, we shall model some of the various execution semantics of ECA rules using the interaction of the execution semantics of transaction programs written in (Con)GOLOG — via the Do predicate — with the relational theories developed so far for database transactions.

$$\begin{array}{l} <trans : Update\_stocks : price\_inserted : \\ \quad (\exists c, time, bal, price')[price\_inserted(s\_id, price, time) \wedge customer(c, bal, s\_id) \wedge \\ \qquad\qquad stock(s\_id, price', clos\_pr)] \\ \quad \longrightarrow \\ \quad stock\_insert(s\_id, price, clos\_pr) > \end{array}$$

$$\begin{array}{l} <trans : Buy\_100shares : price\_inserted : \\ \quad (\exists\ new\_price, time, bal, pr, clos\_pr)[price\_inserted(s\_id, new\_price, time) \wedge \\ \qquad customer(c, bal, s\_id) \wedge stock(s\_id, pr, clos\_pr) \wedge new\_price < 50 \wedge clos\_pr > 70] \\ \quad \longrightarrow \\ \quad buy(c, s\_id, 100) > \end{array}$$

Fig. 1. Rules for updating stocks and buying shares.

4.2 Rule Language

An *ECA rule* is a construct of the following form:

$$<t : R : \tau : \zeta(\boldsymbol{x}) \longrightarrow \alpha(\boldsymbol{y})> . \tag{12}$$

In this construct, t specifies the transaction that fires the rule, τ specifies events that trigger the rule, and R is a constant giving the rule's identification number (or name). A rule is triggered if the event specified in its event part occurred since the beginning of the open transaction in which that event part is evaluated. Events are one of the predicates $F\_inserted(r, t, s)$ and $F\_deleted(r, t, s)$, or a combination thereof using logical connectives. The ζ part specifies the rule's condition; it mentions predicates $F\_inserted(r, \boldsymbol{x}, t, s)$ and $F\_deleted(r, \boldsymbol{x}, t, s)$ called *transition fluents*, which denote the transition tables [29] corresponding to insertions into and deletions from the relation F. In (12), arguments t, R, and s are suppressed from all the fluents; the first two ones are restored when (12) is translated to a GOLOG program, and s is restored at run time. Finally, α gives a GOLOG program which will be executed upon the triggering of the rule once the specified condition holds. Actions also may mention transition fluents. Notice that $\boldsymbol{x}$ are free variables mentioned by ζ and contain all the free variables $\boldsymbol{y}$ mentioned by α.

Example 2. Consider the following active behavior for Example 1. For each customer, his stock is updated whenever new prices are notified. When current prices are being updated, the closing price is also updated if the current notification is the last of the day; moreover, suitable trade actions are initiated if some conditions become true of the stock prices, under the constraint that balances cannot drop below a certain amount of money. Two rules for our example are shown in Figure 1. □

To characterize the notion of transition tables and events, we introduce the fluent $considered(r, t, s)$ which intuitively means that the rule r can be considered for execution in situation s with respect to the transaction t. The following gives an abbreviation for $considered(r, t, s)$:

$$considered(r,t,s) =_{df} (\exists t').running(t',s). \tag{13}$$

Intuitively, this means that, as long as t is running, any rule r may be considered for execution. In actual systems this concept is more sophisticated than this scheme[6].

For each database fluent $F(\boldsymbol{x},t,s)$, we introduce two special fluents called *transition fluents*, namely $F\_inserted(r,\boldsymbol{x},t,s)$ and $F\_deleted(r,\boldsymbol{x},t,s)$. The following successor state axiom characterizes $F\_inserted(r,\boldsymbol{x},t,s)$:

$$\begin{aligned} F\_inserted(r,\boldsymbol{x},t,do(a,s)) \equiv & considered(r,t,s) \wedge a = F\_insert(\boldsymbol{x},t)] \vee \\ & F\_inserted(r,\boldsymbol{x},t,s) \wedge \neg a = F\_delete(\boldsymbol{x},t). \end{aligned} \tag{14}$$

Definition (14) means that a tuple $\boldsymbol{x}$ is considered inserted in situation $do(a,s)$ iff the internal action $F\_insert(\boldsymbol{x},t)$ was executed in the situation s while the rule r was considered, or it was already inserted and a is not the internal action $F\_delete(\boldsymbol{x},t)$. This captures the notion of *net effects* [29] of a sequence of actions. Such net effects are accumulating only changes that really affect the database; in this case, if a record is deleted after being inserted, this amounts to nothing having happened. Further net effect policies can be captured in this axiom. The transition fluent $F\_deleted(r,\boldsymbol{x},t,s)$ is characterized in a similar way.

Finally, for each database fluent $F(\boldsymbol{x},t,s)$, we introduce the following two further special fluents called *event fluents*: $F\_inserted(r,t,s)$, and $F\_deleted(r,t,s)$. The event fluent $F\_inserted(r,t,s)$ corresponding to an insertion into the relation F has the following successor state axiom:

$$\begin{aligned} & F\_inserted(r,t,do(a,s)) \equiv \\ & \quad a = F\_insert(\boldsymbol{x},t') \wedge considered(r,t,s) \vee F\_inserted(r,t,s). \end{aligned} \tag{15}$$

The event fluent $F\_deleted(r,t,s)$ corresponding to a deletion from the relation F has a similar successor state axiom.

4.3 Active Relational Theories

An *active relational language* is a relational language extended in the following way: for each $n+2$-ary fluent $F(\boldsymbol{x},t,s)$, we introduce two $n+3$-ary transition fluents of the form $F\_inserted(r,\boldsymbol{x},t,s)$ and $F\_deleted(r,\boldsymbol{x},t,s)$, and two 3-ary event fluents of the form $F\_inserted(r,t,s)$ and $F\_deleted(r,t,s)$.

Definition 4. (Active Relational Theory for flat transactions) *A theory $\mathcal{D} \subseteq \mathfrak{W}$ is an* active relational theory *iff it is of the form*

$$\mathcal{D} = \mathcal{D}_{brt} \cup \mathcal{D}_{tf} \cup \mathcal{D}_{ef},$$

where

1. *$\mathcal{D}_{brt}$ is a basic relational theory.*
2. *$\mathcal{D}_{tf}$ is the set of axioms for transition fluents.*
3. *$\mathcal{D}_{ef}$ is the set of axioms for event fluents which capture simple events. Complex events are defined by combining the event fluents using $\neg$ and $\wedge$.*

[6] For example, in Starburst [29], r will be considered in the future course of actions only from the time point where it last stopped being considered.

4.4 Execution Models

In this section, we specify the execution models of active databases by assuming that the underlying transaction model is that of flat transactions.

Classification. The three components of the ECA rule — i.e. **E**vent, **C**ondition, and **A**ction — are the main dimensions of the representational component of active behavior. Normally, either an indication is given in the rule language as to how the ECA rules are to be processed, or a given processing model is assumed by default for all the rules of the ADBMS. To ease our presentation, we assume the execution models by default.

An execution model is tightly related to the coupling modes specifying the timing constraints of the evaluation of the rule's condition and the execution of the rule's action relative to the occurrence of the event that triggers the rule. We consider the following coupling modes[7]:

1. EC coupling modes:
   Immediate: Evaluate C immediately after the ECA rule is triggered.
   Delayed: Evaluate C at some delayed time point, usually after having performed many other database operations since the time point at which the rule has been triggered.
2. CA coupling modes:
   Immediate: Execute A immediately after C has been evaluated.
   Delayed: Execute A at some delayed time point, usually after having performed many other database operations since the time point at which C has been evaluated.

The execution model is also tightly related to the concept of transaction. In fact, the question of determining when to process the different components of an ECA rule is also answered by determining the transactions within which – if any – the C and A components of the ECA rule are evaluated and executed, respectively. In other words, the transaction semantics offer the means for controlling the coupling modes by allowing one the flexibility of processing the rule components in different, well-chosen transactions. In the sequel, the transaction triggering a rule will be called *triggering transaction* and any other transaction launched by the triggering transaction will be called *triggered transaction*. We assume that all database operations are executed within the boundaries of transactions. From this point of view, we obtain the following refinement for the delayed coupling mode:

1. **Delayed** EC coupling mode: Evaluate C at the end of the triggering transaction T, after having performed all the other database operations of T, but before T's terminal action.
2. **Delayed** CA coupling mode: Execute A at the end of the triggering transaction T, after having performed all the other database operations of T and after having evaluated C, but before T's terminal action.

In presence of flat transactions, we also obtain the following refinement of the immediate coupling mode:

[7] Our presentation is slightly more general than the original one in [14], in which the relationships between coupling modes and execution models, and those between transactions and execution models were not conceptually separated.

1. **Immediate** EC coupling mode: Evaluate C within the triggering transaction immediately after the ECA rule is triggered.
2. **Immediate** CA coupling mode: Execute A within the triggering transaction immediately after evaluating C.

Notice that the semantics of flat transactions rules out the possibility of nested transactions. For example, we can not process C in a flat transaction and then process A in a further flat transaction, since we quickly encounter the necessity of nesting transactions whenever the execution of a rule triggers further rules. Also, we can not have a delayed CA coupling mode such as: Execute A at the end of the triggering transaction T in a triggered transaction T', after having performed all the other database operations of T, after T's terminal action, and after the evaluation of C. The reason is that, in the absence of nesting of transactions, we will end up with a large set of flat transactions which are independent from each other. This would make it difficult to relate these independent flat transactions as belonging to the processing of a few single rules.

The refinements above yield for each of the EC and CA coupling modes two possibilities: (1) immediate, and (2) delayed. There are exactly 4 combinations of these modes. We will denote these combinations by pairs (i, j) where i and j denote an EC and a CA coupling modes, respectively. For example, $(1, 2)$ is a coupling mode meaning a combination of the immediate EC and delayed CA coupling modes. Moreover, we will call the pairs (i, j) interchangeably coupling modes or execution models. The context will be clear enough to determine what we are writing about. However, we have to consider these combinations with respect to the constraint that we always execute A strictly after C is evaluated[8]. The following combinations satisfy this constraint: $(1, 1)$, $(1, 2)$, and $(2, 2)$; the combination $(2, 1)$, on the contrary, does not satisfy the constraint.

Immediate Execution Model. Here, we specify the execution model $(1, 1)$. This can be formulated as: **Evaluate C immediately after the ECA rule is triggered and execute A immediately after evaluating C within the triggering transaction.**

Suppose we have a set $\mathcal{R}$ of n ECA rules of the form (12). Then the following GOLOG procedure captures the immediate execution model $(1, 1)$:

$$
\begin{array}{l}
\textbf{proc}\ Rules(t) \\
(\pi \boldsymbol{x}_1, \boldsymbol{y}_1)[\tau_1[R_1, t]? \ ; \ \zeta_1(\boldsymbol{x}_1)[R_1, t]? \ ; \ \alpha_1(\boldsymbol{y}_1)[R_1, t]]| \\
\qquad\qquad\qquad\qquad\vdots \\
(\pi \boldsymbol{x}_n, \boldsymbol{y}_n)[\tau_n[R_n, t]? \ ; \ \zeta_n(\boldsymbol{x}_n)[R_n, t]? \ ; \ \alpha_n(\boldsymbol{y}_n)[R_n, t]]| \\
\neg[(\exists \boldsymbol{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\boldsymbol{x})[R_1, t]) \vee \ldots \\
\qquad\qquad\qquad\qquad\qquad \vee\ (\exists \boldsymbol{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\boldsymbol{x}_n)[R_n, t])]\ ? \\
\ \textbf{endProc}\ .
\end{array}
\tag{16}
$$

[8] This constraint is in fact stricter than a similar constraint found in [14], where it is stated that "A cannot be executed before C is evaluated". The formulation of [14], however, does not rule out simultaneous action executions and condition evaluations, a situation that obviously can lead to disastrous behaviors.

The notation $\tau[r, t]$ means the result of restoring the arguments r and t to all event fluents mentioned by τ, and $\zeta(\boldsymbol{x})[r, t]$ means the result of restoring the arguments r and t to all transition fluents mentioned by ζ. For example, if τ is the complex event

$$price\_inserted \wedge customer\_inserted,$$

then $\tau[r, t]$ is

$$price\_inserted(r, t) \wedge customer\_inserted(r, t).$$

Notice that the procedure (16) above formalizes *how* rules are processed using the immediate model examined here: the procedure $Rules(t)$ nondeterministically selects a rule R_i (hence the use of |), tests if an event $\tau_i[R_i, t]$ occurred (hence the use of ?), in which case it immediately tests whether the condition $\zeta_i(\boldsymbol{x}_i)[R_i, t]$ holds (hence the use of ;), at which point the action part $\alpha_i(\boldsymbol{y}_i)$ is executed. The last test condition of (16) permits to exit from the rule procedure when none of the rules is triggered.

Delayed Execution Model. Now, we specify the execution model $(2, 2)$ that has both EC and CA coupling being delayed modes. This asks to **evaluate C and execute A at the end of a transaction between the transaction's last action and either its commitment or its failure**. Notice that A must be executed after C has been evaluated.

The following GOLOG procedure captures the delayed execution model $(2, 2)$:

$$\begin{array}{l}
\textbf{proc } Rules(t) \\
(\pi \boldsymbol{x}_1, \boldsymbol{y}_1)[\tau_1[R_1, t]? \;;\; (\zeta_1(\boldsymbol{x}_1)[R_1, t] \wedge assertionInterval(t))? \;;\; \alpha_1(\boldsymbol{y}_1)]| \\
\qquad\qquad\qquad\qquad\vdots \\
(\pi \boldsymbol{x}_n, \boldsymbol{y}_n)[\tau_n[R_n, t]? \;;\; (\zeta_n(\boldsymbol{x}_n)[r_n, t] \wedge assertionInterval(t))? \;;\; \alpha_n(\boldsymbol{y}_n)]| \\
\neg\{[(\exists \boldsymbol{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\boldsymbol{x})[R_1, t]) \vee \ldots \\
\qquad\qquad \vee (\exists \boldsymbol{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\boldsymbol{x}_n)[R_n, t])] \wedge assertionInterval(t)\}\; ? \\
\textbf{endProc}.
\end{array} \tag{17}$$

Let the interval between the end of a transaction (i.e., the situation $do(End(t), s)$, for some s) and its termination (i.e., the situation $do(Commit(t), s)$ or $do(Rollback(t), s)$, for some s) be called *assertion interval*. The fluent $assertionInterval(t, s)$ used in (17) captures this notion. The following successor state axiom characterizes this fluent:

$$\begin{array}{c}
assertionInterval(t, do(a, s)) \equiv a = End(t) \vee \\
assertionInterval(t, s) \wedge \neg termAct(a, t).
\end{array} \tag{18}$$

In (17), both the C and A components of triggered rules are executed at assertion intervals.

Mixed Execution Model. Here, we specify the execution model $(1, 2)$ that mix both immediate EC and delayed CA coupling modes. This execution model asks to **evaluate C immediately after the ECA rule is triggered and to execute A after evaluating C in the assertion interval.** This model has the semantics

$$\begin{array}{l}
\textbf{proc } Rules(t) \\
(\pi \boldsymbol{x}_1, \boldsymbol{y}_1)[\tau_1[R_1, t]? \; ; \; \zeta_1(\boldsymbol{x}_1)[R_1, t]? \; ; \; assertionInterval(t)? \; ; \; \alpha_1(\boldsymbol{y}_1)] | \\
\qquad\qquad\qquad \vdots \\
(\pi \boldsymbol{x}_n, \boldsymbol{y}_n)[\tau_n[R_n, t]? \; ; \; \zeta_n(\boldsymbol{x}_n)[r_n, t]? \; ; \; assertionInterval(t))? \; ; \; \alpha_n(\boldsymbol{y}_n)] | \\
\neg\{[(\exists \boldsymbol{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\boldsymbol{x})[R_1, t]) \vee \ldots \\
\qquad \vee (\exists \boldsymbol{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\boldsymbol{x}_n)[R_n, t])] \wedge assertionInterval(t)\} \; ? \\
\textbf{endProc}.
\end{array} \tag{19}$$

Here, only the A components of triggered rules are executed at assertion intervals.

Abstract Execution of Rule Programs. We "run" a GOLOG program T embodying an active behavior by establishing that

$$\mathcal{D} \models (\exists s) \; Do(T, S_0, s), \tag{20}$$

where S_0 is the initial, empty log, and $\mathcal{D}$ is the active relational theory for flat transactions. This exactly means that we look for some log that is generated by the program T, and pick any instance of s resulting from the proof obtained by establishing this entailment.

5 Classification Theorems

There is a natural question which arises with respect to the different execution models whose semantics have been given above: what (logical) relationship may exist among them? To answer this question, we must develop a (logical) notion of equivalence between two given execution models. Suppose that we are given two programs $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ corresponding to the execution models (i, j) and (k, l), respectively.

Definition 5. (Database versus system queries) *Suppose Q is a situation calculus query. Then Q is a* database query *iff the only fluents it mentions are database fluents. A* system query *is one that mentions at least one system fluent.*

Establishing an equivalence between the programs $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ with respect to an active relational theory $\mathcal{D}$ amounts to establishing that, for all database queries $Q(s)$ and transactions t, whenever the answer to $Q(s)$ is "yes" in a situation resulting from the execution of $Rules^{(i,j)}(t)$ in S_0, executing $Rules^{(k,l)}(t)$ in S_0 results in a situation yielding "yes" to $Q(s)$.

Definition 6. (Implication of Execution Models) *Suppose $\mathcal{D}$ is an active relational theory, and let $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ be ConGOLOG programs corresponding to the execution models (i, j) and (k, l), respectively. Moreover, suppose that for all database queries Q, we have*

$$(\forall s, s', s'', t).Do(Rules^{(m,n)}(t), s, s') \wedge Do(Rules^{(m,n)}(t), s, s'') \supset Q[s'] \equiv Q[s''],$$

where (m, n) is (i, j) or (k, l). Then a rule program $Rules^{(i,j)}(t)$ implies another rule program $Rules^{(k,l)}(t)$ ($Rules^{(i,j)}(t) \Longrightarrow Rules^{(k,l)}(t)$) iff, for every database query Q,

$$\begin{aligned}(\forall t, s)\{[(\exists s').Do(Rules^{(i,j)}(t), s, s') \wedge Q[s']] \supset \\ [(\exists s'').Do(Rules^{(k,l)}(t), s, s'') \wedge Q[s'']]\}.\end{aligned} \tag{21}$$

Definition 7. (Equivalence of execution models) *Assume the conditions and notations of Definition 6. Then $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ are equivalent ($Rules^{(i,j)}(t) \cong Rules^{(k,l)}(t)$) iff, for every database query Q,*

$$\begin{aligned}(\forall t, s)\{[(\exists s').Do(Rules^{(i,j)}(t), s, s') \wedge Q[s']] \equiv \\ [(\exists s'').Do(Rules^{(k,l)}(t), s, s'') \wedge Q[s''])]\}.\end{aligned}$$

We restrict our attention to database queries since we are interested in the final state of the content of the database, regardless of the final values of the system fluents.

Theorem 1. *Assume the conditions of Definition 6. Then $Rules^{(2,2)}(t) \Longrightarrow Rules^{(1,1)}(t)$.*

Proof (Outline). By Definition 6, we must prove that, whenever Q is a database query, we have

$$\begin{aligned}(\forall t, s).[(\exists s').Do(Rules^{(2,2)}(t), s, s') \wedge Q[s']] \supset \\ [(\exists s'').Do(Rules^{(1,1)}(t), s, s'') \wedge Q[s'']].\end{aligned} \tag{22}$$

Therefore, by the definitions of $Rules^{(1,1)}(t)$, $Rules^{(2,2)}(t)$, *Do*/3, and the semantics of ConGolog given in [8], we must prove:

$$\begin{aligned}
&(\forall t, s). \\
&\{(\exists s').Trans^*(\{(\pi \boldsymbol{x}_1, \boldsymbol{y}_1)[\tau_1[R_1, t]? \; ; \; (\zeta_1(\boldsymbol{x}_1)[R_1, t] \wedge \\
&\qquad\qquad assertionInterval(t))? \; ; \; \alpha_1(\boldsymbol{y}_1)]| \\
&\qquad\qquad\qquad \vdots \\
&\quad (\pi \boldsymbol{x}_n, \boldsymbol{y}_n)[\tau_n[R_n, t]? \; ; \; (\zeta_n(\boldsymbol{x}_n)[r_n, t] \wedge \\
&\qquad\qquad assertionInterval(t))? \; ; \; \alpha_n(\boldsymbol{y}_n)]| \\
&\quad \neg\{[(\exists \boldsymbol{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\boldsymbol{x})[R_1, t]) \vee \ldots \vee (\exists \boldsymbol{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\boldsymbol{x}_n)[R_n, t])] \wedge \\
&\qquad assertionInterval(t)\} \, ?\}, \; s, \; nil, \; s') \wedge Q[s']\} \supset \\
&\{(\exists s'').Trans^*(\{(\pi \boldsymbol{x}_1, \boldsymbol{y}_1)[\tau_1[R_1, t]? \; ; \; \zeta_1(\boldsymbol{x}_1)[R_1, t]? \; ; \; \alpha_1(\boldsymbol{y}_1)[R_1, t]]| \\
&\qquad\qquad\qquad \vdots \\
&\qquad (\pi \boldsymbol{x}_n, \boldsymbol{y}_n)[\tau_n[R_n, t]? \; ; \; \zeta_n(\boldsymbol{x}_n)[R_n, t]? \; ; \; \alpha_n(\boldsymbol{y}_n)[R_n, t]]| \\
&\qquad \neg[(\exists \boldsymbol{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\boldsymbol{x})[R_1, t]) \vee \ldots \vee (\exists \boldsymbol{x}_n)(\tau_n[R_n, t] \wedge \\
&\qquad\qquad \zeta_n(\boldsymbol{x}_n)[R_n, t])] \, ?\}, \; s, \; nil, \; s'') \wedge Q[s'']\}.
\end{aligned}$$

By the semantics of $Trans$, we may unwind the $Trans^*$ predicate in the antecedent of the big formula above to obtain, in each step, a formula which is a big disjunction of the form

$$\begin{aligned}&(\exists s').[(\phi_1^1 \land \phi_2^1 \land assertionInterval(t) \land \phi_3^1) \lor \cdots \\ &\quad \lor (\phi_1^n \land \phi_2^n \land assertionInterval(t) \land \phi_3^n) \lor \\ &\qquad \Phi] \land Q[s'],\end{aligned} \tag{23}$$

where ϕ_1^i represents the formula $\tau_i[R_i, t]$, ϕ_2^i represents $\zeta_i(\boldsymbol{x}_i)[R_i, t]$, and ϕ_3^i represents the situation calculus formula generated from $\alpha_i(\boldsymbol{y}_i)$, with $i = 1, \cdots, n$; Φ represents the formula in the last test action of $Rules^{(2,2)}(t)$. Similarly, we may unwind the $Trans^*$ predicate in the consequent of the big formula on the previous page, second column, to obtain, in each step, a formula which is a big disjunction of the form

$$\begin{aligned}&(\exists s'').[(\phi_1^1 \land \phi_2^1 \land \phi_3^1) \lor \\ &\cdots \qquad \lor (\phi_1^n \land \phi_2^n \land \phi_3^n) \lor \Phi'] \land Q[s''],\end{aligned} \tag{24}$$

where ϕ_1^i, ϕ_2^i, and ϕ_3^i are to interpret as above, and Φ' represents the formula in the last test action of $Rules^{(1,1)}(t)$. Φ' differs from Φ only through the fact that Φ is a conjuction with one more conjunct which is $assertionInterval(t)$. Also, since no nested transaction is involved, and since both rule programs involved are confluent, we may set $s' = s''$. Therefore, clearly (23) implies (24). ■

Theorem 2. *Assume the conditions of Definition 6. Then* $Rules^{(1,2)}(t) \cong Rules^{(2,2)}(t)$.

Proof. This proof is similar to that of Theorem 1, so we omit it. ■

Corollary 1. *Assume the conditions of Definition 6. Then* $Rules^{(1,2)}(t) \Longrightarrow Rules^{(1,1)}(t)$.

Proof. The proof is immediate from Theorems 1 and 2. ■

Contrary to what we did in this section, where we assume flat transactions, we could also specify the execution models of active databases by assuming that the underlying transaction model is that of nested transactions [23]. In the latter case, various flavors of immediate and delayed execution models are considered. They all are shown to imply the basic case $Rules^{(1,1)}$. Moreover, all the execution models examined are not equivalent to $Rules^{(1,1)}$ [16].

6 Discussion

Among logic-oriented researchers [31, 19, 9, 1, 2], Baral and Lobo develop a situation calculus-based language to describe actions and their effects, events, and evaluation modes of active rules [2]. In [4], Bertossi *et al.* propose a situation calculus-based formalization that differ considerably from our approach. Their representation of rules forces the action part of their rules to be a primitive database operation. Unlike Bertossi

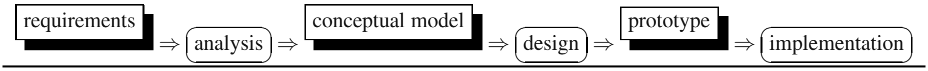


Fig. 2. Relational theories as conceptual models of active database management systems.

et al., we base our approach on GOLOG, which allows the action component of our rules to be arbitrary GOLOG programs. Moreover, transactions are first-class citizens in their approach.

Thus far, ACTA [6] seems to our knowledge the only framework addressing the problem of specifying transaction models in full first order logic. We complement the approach of ACTA with a formalization of rules and give implementable specifications. The method for such an implementation is given elsewhere [16].

We have used one single logic – the situation calculus — to accounts for virtually all features of rule languages and execution models of ADBMSs. The output of this account is a conceptual model for ADBMSs in the form of active relational theories to be used in conjunction with a theory of complex actions. Thus, considering the software development cycle as depicted in Figure 2, an active relational theory corresponding to an ADBMS constitutes, together with a theory of complex actions a conceptual model for that ADBMS. Since our theories are implementable specifications, implementing the conceptual model provides one with a rapid prototype of the specified ADBMS.

The emerging SQL3 standard [17] contains further dimensions of active behavior that are important and are not mentioned in this paper:

- AFTER/BEFORE/INSTEAD rule activation: SQL3 provides the possibility that a rule be fired before, after, or instead of a certain event.
- Rule activation granularity: rules can be activated for each row or for each statement.
- Interruptability: rule actions may be interruptable in order for other triggered rules to be activated or not.

These issues are dealt with in [16].

Ideas expressed here may be extended in various ways. First, real ADBMSs may be given the same kind of semantics to make them comparable with respect to a uniform framework; [16] addresses this issue. Second, formalizing rules as GOLOG programs can be fruitful in terms of proving formal properties of active rules since such properties can be proved as properties of GOLOG programs. Here, the problems arising classically in the context of active database like confluence and termination [29] are dealt with, and the relationships between the various execution models in terms of their equivalence/nonequivalence are also studied here. This is a reasoning task that will appear in the full version of this paper. Finally, possible rule processing semantics different from existing ones may be studied within our framework.

In Memoriam

Ray Reiter passed away on September 16, 2002 when this work and some others were almost completed. Ray's work has been very influential in foundations of databases and artificial intelligence. To pick just a few of his most important contributions, his concept of Closed World Assumption in database theory, his Default Logic, and his formalization of system diagnosis are well known. His last major contribution was in the area of specifications using the situation calculus [28]. Many in the field considered Ray to be an exceptionally intelligent, and at the same time a very humble human being. The deep intellectual impression that he left and his great achievements will last for years to come. It is hard to imagine that Ray is no longer with us; we miss him very much. Salut Ray!

References

1. C. Baral and J. Lobo. Formal characterizations of active databases. In *International Workshop on Logic in Databases, LIDS'96*, 1996.
2. C. Baral, J. Lobo, and G. Trajcevski. Formal characterizations of active databases: Part ii. In *Proceedings of Deductive and Object-Oriented Databases, DOOD'97*, 1997.
3. P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, Reading, MA, 1987.
4. L. Bertossi, J. Pinto, and R. Valdivia. Specifying database transactions and active rules in the situation calculus. In H. Levesque and F. Pirri, editors, *Logical Foundations of Cognitive Agents. Contributions in Honor of Ray Reiter*, pages 41–56, New-York, 1999. Springer Verlag.
5. A. Bonner and M. Kifer. Transaction logic programming. Technical report, University of Toronto, 1992.
6. P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.
7. T. Coupaye and C. Collet. Denotational semantics for an active rule execution model. In T. Sellis, editor, *Rules in Database Systems: Proceedings of the Second International Workshop, RIDS '95*, pages 36–50. Springer Verlag, 1995.
8. G. De Giacomo, Y. Lespérance, and H.J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogeneous actions in the situation calculus. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1221–1226, 1997.
9. A.A.A. Fernandes, M.H. Williams, and N.W. Paton. A logic-based integration of active and deductive databases. *New Generation Computing*, 15(2):205–244, 1997.
10. P. Fraternali and L. Tanca. A structured approach to the definition of the semantics of active databases. *ACM Transactions on Database Systems*, 20:414–471, 1995.
11. A. Gabaldon. Non-markovian control in the situation calculus. In G. Lakemeyer, editor, *Proceedings of the Second International Cognitive Robotics Workshop*, pages 28–33, Berlin, 2000.
12. J. Gray and Reuter A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1995.
13. A. Guessoum and J.W. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, 1990.
14. M. Hsu, R. Ladin, and R. McCarthy. An execution model for active database management systems. In *Proceedings of the third International Conference on Data and Knowledge Bases*, pages 171–179. Morgan Kaufmann, 1988.
15. I. Kiringa. Towards a theory of advanced transaction models in the situation calculus (extended abstract). In *Proceedings of the VLDB 8th International Workshop on Knowledge Representation Meets Databases (KRDB'01)*, 2001.

16. I. Kiringa. *Logical Foundations of Active Databases*. PhD thesis, Computer Science, University of Toronto, Toronto, 2003.
17. K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in sql-3. In N. Paton, editor, *Active Rules in Database Systems*, pages 197–219. Springer Verlag, 1999.
18. H. Levesque, R. Reiter, Y. Lespérance, Fangzhen Lin, and R.B. Scherl. Golog: A logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions*, 31(1-3):59–83, 1997.
19. B. Ludäscher, U. Hamann, and G. Lausen. A logical framework for active rules. In *Proceedings of the Seventh International Conference on Management of Data*, Pune, 1995. Tata and McGraw-Hill.
20. N. Lynch, M.M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, 1994.
21. J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.
22. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.
23. J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing. Information Systems Series*. The MIT Press, Cambridge, MA, 1985.
24. P. Picouet and V. Vianu. Expressiveness and complexity active databases. In *ICDT'97*, 1997.
25. F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *J. ACM*, 46(3):325–364, 1999.
26. R. Reiter. Towards a logical reconstruction of relational database theory. In M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling*, pages 163–189, New-York, 1984. Springer Verlag.
27. R. Reiter. On specifying database updates. *J. of Logic Programming*, 25:25–91, 1995.
28. R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, Cambridge, 2001.
29. J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.
30. M. Winslett. *Updating Logical Databases*. Cambridge University Press, Cambridge, MA, 1990.
31. C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In T.W. Ling and A.O. Mendelzon, editors, *Fourth International Conference on Deductive and Object-Oriented Databases*, pages 55–72, Berlin, 1995. Springer Verlag.

Modelling Dynamic Web Data

Philippa Gardner and Sergio Maffeis[*]

Imperial College London
South Kensington Campus
SW7 2AZ London
United Kingdom
{pg,maffeis}@doc.ic.ac.uk

Abstract. We introduce X$d\pi$, a peer-to-peer model for reasoning about the dynamic behaviour of web data. It is based on an idealised model of semi-structured data, and an extension of the π-calculus with process mobility and with an operation for interacting with data. Our model can be used to reason about behaviour found in, for example, dynamic web page programming, applet interaction, and service orchestration. We study behavioural equivalences for X$d\pi$, motivated by examples.

1 Introduction

Web data, such as XML, plays a fundamental rôle in the exchange of information between globally distributed applications. Applications naturally fall into some sort of mediator approach: systems are divided into peers, with mechanisms based on XML for interaction between peers. The development of analysis techniques, languages and tools for web data is by no means straightforward. In particular, although web services allow for interaction between processes and data, direct interaction between processes is not well-supported.

Peer-to-peer data management systems are decentralised distributed systems where each component offers the same set of basic functionalities and acts both as a producer and as a consumer of information. We model systems where each peer consists of an XML data repository and a working space where processes are allowed to run. Our processes can be regarded as agents with a simple set of functionalities; they communicate with each other, query and update the local repository, and migrate to other peers to continue execution. Process definitions can be included in documents[1], and can be selected for execution by other processes. These functionalities are enough to express most of the dynamic behaviour found in web data, such as web services, distributed (and replicated) documents [1], distributed query patterns [2], hyperlinks, forms, and scripting.

In this paper we introduce the X$d\pi$-calculus, which provides a formal semantics for the systems described above. It is based on a network of locations

[*] Contact author. Tel: +44 (0)20 7594 8180. Supported by a grant from Microsoft Research, Cambridge.

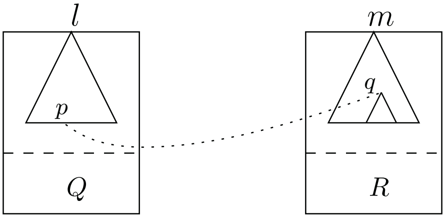
[1] We regard process definitions in documents as an atomic piece of data, and we do not consider queries which modify such definitions.

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 130–146, 2004.

(peers) containing a (semi-structured) data model, and π-like processes [3–5] for modeling process interaction, process migration, and interaction with data. The data model consists of unordered labelled trees, with embedded processes for querying and updating such data, and explicit pointers for referring to other parts of the network: for example, a document with a hyperlink referring to another site and a light-weight trusted process for retrieving information associated with the link. The idea of embedding processes (scripts) in web data is not new: examples include Javascript, SmartTags and calls to web services. However, web applications do not in general provide direct communication between active processes, and process coordination therefore requires specialised orchestration tools. In contrast, distributed process interaction (communication and co-ordination) is central to our model, and is inspired by the current research on distributed process calculi.

We study behavioural equivalences for X$d\pi$. In particular, we define when two processes are equivalent in such a way that when the processes are put in the same position in a network, the resulting networks are equivalent. We do this in several stages. First, we define what it means for two X$d\pi$-networks to be equivalent. Second, we indicate how to translate X$d\pi$ into a simpler calculus (Xπ_2), where the location structure has been pushed inside the data and processes. This translation technique, first proposed in [6], enables us to separate reasoning about processes from reasoning about data and networks. Finally, we define process equivalence and study examples. In particular, we indicate how to prove that services can be transparently replicated, using a proof method based on a labelled transition system for the Xπ_2-calculus. Full details on the translation and equivalences can be found in [7].

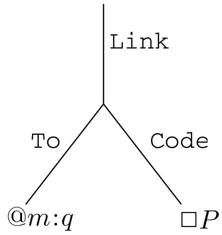
A Simple Example. We use the *hyperlink example* as a simple example to illustrate our ideas. Consider two locations (machines identified by their IP addresses) l and m. Location l contains a hyperlink at p referring to data identified by path q at location m:



In the working space of location l, the process Q activates the process embedded in the hyperlink, which then fires a request to m to copy the tree identified by path q and write the result to p at l.

The hyperlink at l, written in both XML notation (LHS) and ambient notation used in this paper (RHS), is:

```
<Link>
    <To>
     m:q
    </To>
    <Code>
     P
    </Code>
</Link>
```



$$\mathtt{Link}[\,\mathtt{To}[\,@m{:}q\,]\,|\,\mathtt{Code}[\,\Box P\,]\,]$$

This hyperlink consists of two parts: an external pointer $@m{:}q$, and a scripted process $\Box P$ which provides the mechanism to fetch the subtree q from m. Process P has the form

$$P = \mathsf{read}_{p/\mathtt{Link}/\mathtt{To}}(@x{:}y).\overline{load}\langle x, y, p\rangle$$

The read command reads the pointer reference from position $p/\mathtt{Link}/\mathtt{To}$ in the tree, substitutes the values m and q for the parameters x and y in the continuation and evolves to the output process $\overline{load}\langle m, q, p\rangle$, which records the target location m, the target path q and the position p where the result tree will go. This output process calls a corresponding input process inside Q, using π-calculus interaction. The specification of Q has the form:

$$Q_s \;= !load(x, y, z).\mathsf{go}\,x.\,\mathsf{copy}_y(X).\mathsf{go}\,l.\mathsf{paste}_z\langle X\rangle$$

The replication ! denotes that the input process can be used as many times as requested. The interaction between the $\overline{load}$ and $load$ replaces the parameters x, y, z with the values m, q, p in the continuation. The process then goes to m, copies the tree at q, comes back to l, and pastes the tree to p.

The process Q_s is just a specification. We refine this specification by having a process Q (acting as a service call), which sends a request to location m for the tree at q, and a process R (the service definition) which grants the request. Processes Q and R are defined by

$$Q = !load(x, y, z).(\nu c)(\mathsf{go}\,x.\overline{get}\langle y, l, c\rangle \mid c(X).\mathsf{paste}_z\langle X\rangle)$$

$$R = !get(y, x, w).\mathsf{copy}_y(X).\mathsf{go}\,x.\overline{w}\langle X\rangle$$

Once process Q receives parameters from $\overline{load}$, it splits into two parts: the process that sends the output message $\overline{get}\langle q, l, c\rangle$ to m, with information about the particular target path q, the return location l and a private channel name c (created using the π-calculus restriction operator ν), and the process $c(X).\mathsf{paste}_p\langle X\rangle$ waiting to paste the result delivered via the unique channel c. Process R receives the parameters from $\overline{get}$ and returns the tree to the unique channel c at l. Using our definition of process equivalence, we show that Q does indeed have the same intended behaviour as its specification Q_s, and that the processes are interchangeable independently from the context where they are installed.

Related Work. Our work is related to the Active XML approach to data integration developed independently by Abiteboul et al. [8]. They introduce an architecture which is a peer-to-peer system where each peer contains a data model very similar to ours (but where only service calls can be scripted in documents), and a working space where only web service definitions are allowed. Moreover Active XML service definitions cannot be moved around. In this respect our approach is more flexible: for example, we can define an auditing process for assessing a university course—it goes to a government site, selects the assessment criteria appropriate for the particular course under consideration, then moves this information (web service definition) to the university to make the assessment.

Several distributed query languages, such as [9–11], extend traditional query languages with facilities for distribution awareness. Our approach is closest to the one of Sahuguet and Tannen [2], who introduce the ubQL query language based on streams for exchanging large amounts of distributed data, partly motivated by ideas from the π-calculus. There has been much study of data models for the web in the XML, database and process algebra communities. Our ideas have evolved from those found in [12] and [13]. Our process-algebra techniques have most in common with [14, 6, 15]. Process calculi have also been used for example to study security properties of web services [16], reason about mobile resources [17], and in [18] to sketch a distributed query language. Bierman and Sewell [19] have recently extended a small functional language for XML with π-calculus-based concurrency in order to program Home Area Networks devices.

Our work is the first attempt to integrate the study of mobile processes and semi-structured data for Web-based data-sharing applications, and is characterised by its emphasis on dynamic data.

2 A Model for Dynamic Web Data

We model a peer-to-peer system as a sets of interconnected locations (*networks*), where the content of each location consists of an abstraction of a XML data repository (the *tree*) and a term representing both the services provided by the peer and the agents in execution on behalf of other peers (the *process*). Processes can query and update the local data, communicate with each other through named *channels* (public or private), and migrate to other peers. Process definitions can be included in documents and can be selected for execution by other processes.

Trees. Our data model extends the unordered labelled rooted trees of [13], with leaves which can either be scripted processes or pointers to data. We use the following constructs: *edge labels* denoted by $\mathsf{a}, \mathsf{b}, \mathsf{c} \in \mathcal{A}$, *path expressions* denoted by $p, q \in \mathcal{E}$ used to identify specific subtrees, and *locations* of the form $\circlearrowleft, l, m \in \mathcal{L}$, where the 'self' location $\circlearrowleft$ refers to the enclosing location. The set of data trees, denoted $\mathcal{T}$, is given by

$$T ::= 0 \mid T|T \mid \mathsf{a}[T] \mid \mathsf{a}[\Box P] \mid \mathsf{a}[@l{:}p]$$

Tree 0 denotes a rooted tree with no content. Tree $T_1 \mid T_2$ denotes the composition of T_1 and T_2, which simply joins the roots. A tree of the form $\mathsf{a}[\ldots]$ denotes a tree with a single branch labelled a which can have three types of content: a subtree T; a *scripted process* $\Box P$ which is a static process awaiting a command to run; a *pointer* $@l{:}p$ which denotes a pointer to a set of subtrees identified by path expression p in the tree at location l. Processes and path expressions are described below. The structural congruence for trees states that trees are unordered, and scripted processes are identified up to the structural congruence for processes (see Table 1).

Table 1. Structural congruence for X$d\pi$.

Structural congruence is the least congruence satisfying alpha-conversion, the commutative monoidal laws for $(0, |)$ on trees, processes and networks, and the axioms reported below:

(Trees)

$$U \equiv V \Rightarrow \mathsf{a}[\,U\,] \equiv \mathsf{a}[\,V\,]$$

(Values)

$$P \equiv Q \Rightarrow \Box P \equiv \Box Q \qquad v' \equiv w' \wedge \tilde{v} \equiv \tilde{w} \Rightarrow v', \tilde{v} \equiv w', \tilde{w}$$

(Processes)

$$(\nu c)0 \equiv 0 \qquad c \notin fn(P) \Rightarrow P \,|\, (\nu c)Q \equiv (\nu c)(P \,|\, Q) \qquad (\nu c)(\nu c')P \equiv (\nu c')(\nu c)P$$

$$\tilde{v} \equiv \tilde{w} \Rightarrow \overline{c}\langle\tilde{v}\rangle \equiv \overline{c}\langle\tilde{w}\rangle \qquad V \equiv V' \wedge P \equiv Q \Rightarrow \mathsf{update}_p(\chi, V).P \equiv \mathsf{update}_p(\chi, V').Q$$

(Networks)

$$(\nu c)0 \equiv 0 \qquad c \notin fn(N) \Rightarrow N \,|\, (\nu c)M \equiv (\nu c)(N \,|\, M) \qquad (\nu c)(\nu c')N \equiv (\nu c')(\nu c)N$$

$$P \equiv Q \Rightarrow l\langle P\rangle \Rightarrow l\langle Q\rangle \qquad l\left[T \,\|\, (\nu c)P\right] \equiv (\nu c)l\left[T \,\|\, P\right]$$

$$T \equiv S \wedge P \equiv Q \Rightarrow l\left[T \,\|\, P\right] \equiv l\left[S \,\|\, Q\right]$$

Processes. Our processes are based on π-processes extended with an explicit migration primitive between locations, and an operation for interacting directly with data. The π-processes describe the movement of values between channels. Generic variables are x, y, z, channel names or channel variables are a, b, c and values, ranged over by u, v, w, are

$$u ::= T \mid c \mid l \mid p \mid \Box P \mid x$$

We use the notation $\tilde{z}$ and $\tilde{v}$ to denote vectors of variables and values respectively. Identifiers U, V range over scripted processes, pointers and trees. The set of processes, denoted $\mathcal{P}$, is given by

$$P ::= 0 \mid P \,|\, P \mid (\nu c)P \mid \bar{b}\langle\tilde{v}\rangle \mid b(\tilde{z}).P \mid !b(\tilde{z}).P$$
$$\mid \mathsf{go}\, l.P \mid \mathsf{update}_p(\chi, V).P$$

The processes in the first line of the grammar are constructs arising from the π-calculus: the *nil* process 0, the *composition* of processes $P_1 \,|\, P_2$, the *restriction* $(\nu c)P$ which restricts (binds) channel name c in process P, the *output* process $\bar{b}\langle\tilde{v}\rangle$ which denotes a vector of values $\tilde{v}$ waiting to be sent via channel b, the *input* process $b(\tilde{z}).P$ which is waiting to receive values from an output process via channel b to replace the vector of distinct, bound variables $\tilde{z}$ in P, and *replicated input* $!b(\tilde{z}).P$ which spawns off a copy of $b(\tilde{z}).P$ each time one is requested. We assume a simple sorting discipline, to ensure that the number of values sent along a channel matches the number of variables expected to receive those values. Channel names are partitioned into *public* and *session* channels, denoted $\mathcal{C}_P$ and $\mathcal{C}_S$ respectively. Public channels denote those channels that are intended to have the same meaning at each location, such as *finger*, and cannot be restricted. Session channels are used for process interaction, and are not free in the scripted processes occurring in data.

The *migration* primitive $\mathsf{go}\, l.P$ is common in calculi for describing distributed systems; see for example [15]. It enables a process to go to l and become P. An alternative choice would have been to incorporate the location information inside the other process commands: for example using $\overline{l{\cdot}b}\langle\tilde{v}\rangle$ to denote a process which goes to l and interacts via channel b.

The generic *update* command $\mathsf{update}_p(\chi, U).P$ is used to interact with the data trees. The *pattern* χ has the form

$$\chi ::= X \mid @x{:}y \mid \Box X,$$

where X denotes a tree or process variable. Here U can contain variables and must have the same sort as χ. The variables free in χ are bound in U and P. The update command finds all the values U_i given by the path p, pattern-matches these values with χ to obtain the substitution σ_i when it exists. For each successful pattern-matching, it replaces the U_i with $U\sigma_i$ and evolves to $P\sigma_i$. Simple commands can be derived from this update command, including standard copy_p, cut_p and paste_p commands. We can also derive a run_p command, which prescribes, for all scripted processes $\Box P_i$ found at the end of path p, to run P_i in the workspace.

The structural congruence on processes is similar to that given for the π-calculus, and is given in Table 1. Notice that it depends on the structural congruence for trees, since trees can be passed as values.

Networks. We model networks as a composition of *unique* locations, where each location contains a tree and a set of processes. The set of networks, denoted $\mathcal{N}$, is given by

$$N ::= 0 \mid N \,|\, N \mid l\big[T \,\|\, P\big] \mid (\nu c)N \mid l\langle P\rangle$$

The *location* $l\big[T \,\|\, P\big]$ denotes location l containing tree T and process P. It is well-formed when the tree and process are closed, and the tree contains no free session channels. The network composition $N_1 \mid N_2$ is *partial* in that the location names associated with N_1 and N_2 must be disjoint. Communication between locations is modelled by process migration, which we represent explicitly: the process $l\langle P\rangle$ represents a (higher-order) migration message addressed to l and containing process P, which has left its originating location. In the introduction, we saw that a session channel can be shared between processes at different locations. We must therefore lift the restriction to the network level using $(\nu c)N$. Structural congruence for networks is defined in Table 1, and is analogous to that given for processes.

Path Expressions. In the examples of this paper, we just use a very simple subset of XPath expressions [20]. In our examples, "/" denotes path composition and ".", which can appear only inside trees, denotes the path to its enclosing node.

Our semantic model is robust with respect to any choice of mechanism which, given some expression p, identifies a set of nodes in a tree T. We let $p(T)$ denote the tree T where the nodes identified by p are selected, and we represent a selected node by underlining its contents. For example the selected subtrees below are S' and T':

$$T = \mathsf{a}[\,\mathsf{a}[\,S\,] \mid \mathsf{b}\underline{[\,S'\,]} \mid \mathsf{c}\underline{[\,T'\,]}\,]$$

A path expression such as $*/\mathsf{a}$ might select nested subtrees. We give an example:

$$*/\mathsf{a}(T) = \mathsf{a}\underline{[\,\mathsf{a}\underline{[\,S\,]} \mid \mathsf{b}[\,S'\,] \mid \mathsf{c}[\,T'\,]\,]}$$

Reduction and Update Semantics. The reduction relation $\searrow$ describes the movement of processes across locations, the interaction between processes and processes, and the interaction between processes and data. Reduction is closed under network composition, restriction and structural congruence, and it relies on the updating function $\rightsquigarrow$ reported in Table 3. The reduction axioms are given in Table 2.

Table 2. Reduction Semantics.

(Exit)	$m\big[T \,\|\, Q \mid \mathsf{go}\, l.P\big] \searrow m\big[T \,\|\, Q\big] \mid l\langle P\rangle$
(Enter)	$l\big[T \,\|\, Q\big] \mid l\langle P\rangle \searrow l\big[T \,\|\, Q \mid P\big]$
(Com)	$l\big[T \,\|\, \overline{c}\langle\tilde{v}\rangle \mid c(\tilde{z}).P \mid Q\big] \searrow l\big[T \,\|\, P\{\tilde{v}/\tilde{z}\} \mid Q\big]$
(Com!)	$l\big[T \,\|\, \overline{c}\langle\tilde{v}\rangle \mid !c(\tilde{z}).P \mid Q\big] \searrow l\big[T \,\|\, !c(\tilde{z}).P \mid P\{\tilde{v}/\tilde{x}\} \mid Q\big]$
(Update)	$p(T) \rightsquigarrow_{(p,l,\chi,V,P)} T', P' \Rightarrow l\big[T \,\|\, \mathsf{update}_p(\chi, V).P \mid Q\big] \searrow l\big[T' \,\|\, P' \mid Q\big]$

The rules for process movement between locations are inspired by [21]. Rule (Exit) always allows a process $\mathsf{go}\, l.P$ to leave its enclosing location. At the moment, rule (Enter) permits the code of P to be installed at l provided that location l exists[2]. In future work, we intend to associate some security check to this operation. Process interaction (rules (Com) and (!Com)) is inspired by π-calculus interaction. If one process wishes to output on a channel, and another wishes to input on the same channel, then they can react together and transmit some values as part of that reaction.

Table 3. Update Semantics.

$$(\textsc{Zero})\ 0 \rightsquigarrow_{\Theta} 0, 0 \qquad (\textsc{Link})\ @l{:}p \rightsquigarrow_{\Theta} @l{:}p, 0 \qquad (\textsc{Proc})\ \Box Q \rightsquigarrow_{\Theta} \Box Q, 0$$

$$(\textsc{Par})\ \frac{T \rightsquigarrow_{\Theta} T', R \qquad S \rightsquigarrow_{\Theta} S', R'}{T \mid S \rightsquigarrow_{\Theta} T' \mid S', R \mid R'} \qquad (\textsc{Node})\ \frac{U \rightsquigarrow_{\Theta} U', R}{\mathsf{a}[\,U\,] \rightsquigarrow_{\Theta} \mathsf{a}[\,U'\,], R}$$

$$(\textsc{Up})\ \frac{\mathsf{match}(U\{l/\circlearrowleft, p/.\}, \chi) = \sigma \qquad V\sigma \rightsquigarrow_{\Theta} V', R \qquad \Theta = (p, l, \chi, V, P)}{\mathsf{a}\underline{[\,U\,]} \rightsquigarrow_{\Theta} \mathsf{a}[\,V'\,], P\sigma \mid R}$$

The generic (Update) rule provides interaction between processes and data. Using path p it selects for update some subtrees in T, denoted by $p(T)$, and then applies the updating function $\rightsquigarrow$ to $p(T)$ in order to obtain the new tree T' and the continuation process P'. Given a subtree selected by p, the function $\rightsquigarrow$ pattern matches the subtree with pattern χ to obtain substitution σ (when it exists), updates the subtree with $V\sigma$, and creates process $P\sigma$. A formal definition of $\rightsquigarrow$, parameterised by p, l, χ, V, P, is given in Table 3. Rule (Up) deserves some explanation. First of all it substitutes in U any reference to the current location and path with the actual values l and p, denoted $U\{l/\circlearrowleft, p/.\}$. It then matches the result with χ, to obtain substitution σ; when σ exists, it continues updating $V\sigma$, and when we obtain some subtree V' along with a process R, it replaces U with V' and it returns $P\sigma$ in parallel with R.

Remark 1. The ability to select nested nodes introduces a difference between updating the tree in a top-down rather than bottom-up order. In particular the resulting tree is the same, but a different set of processes P is collected. We chose the top-down approach because is bears a closer correspondence with intuition: a copy of P will be created for each update still visible in the final tree outcome. For example,

$$l\left[\,\mathsf{a}[\,\mathsf{b}[\,T\,]\,] \parallel \mathsf{update}_{*}(X, 0).P\,\right] \searrow l\left[\,\mathsf{a}[\,0\,] \parallel P\{\mathsf{b}[\,T\,]/X\}\,\right]$$

whereas, following a bottom-up approach we would have

$$l\left[\,\mathsf{a}[\,\mathsf{b}[\,T\,]\,] \parallel \mathsf{update}_{*}(X, 0).P\,\right] \searrow l\left[\,\mathsf{a}[\,0\,] \parallel P\{\mathsf{b}[\,0\,]/X\} \mid P\{T/X\}\,\right]$$

[2] This feature is peculiar to our calculus, as opposed to e.g. [15], where the existence of the location to be entered is not a precondition to migration. Our choice makes the study of equivalences non-trivial and models an important aspect of *location failure.*

because first we would transform $\mathsf{a}[\,\mathsf{b}[\,T\,]\,]$ in $\mathsf{a}[\,\mathsf{b}[\,0\,]\,]$ generating $P\{T/X\}$, and then transform $\mathsf{a}[\,\mathsf{b}[\,0\,]\,]$ in $\mathsf{a}[\,0\,]$, generating $P\{\mathsf{b}[\,0\,]/X\}$.

Derived Commands. Throughout our examples, we use the derived commands given in Table 4. In particular note that the definition of run is the only case where we allow the instantiation of a process variable.

Table 4. Derived Commands.

$\mathsf{copy}_p(X).P \triangleq \mathsf{update}_p(X,X).P$	copy the tree at p and use it in P
$\mathsf{read}_p(@x{:}y).P \triangleq \mathsf{update}_p(@x{:}y, @x{:}y).P$	read the pointer at p, use its location and path in P
$\mathsf{cut}_p(X).P \triangleq \mathsf{update}_p(X,0).P$	cut the tree at p and use it in p
$\mathsf{paste}_p\langle T\rangle.P \triangleq \mathsf{update}_p(X, X \mid T).P$	where X is not free in T or P, paste tree T at p and evolve to P
$\mathsf{run}_p \triangleq \mathsf{update}_p(\Box X, \Box X).X$	run the scripted process at p

Example 1. The following reaction illustrates the cut command:

$$l\left[\,\mathsf{c}[\,\mathsf{a}[\,T\,] \mid \mathsf{a}[\,T'\,] \mid \mathsf{b}[\,S\,]\,] \,\|\, \mathsf{cut}_{\mathsf{c/a}}(X).P\,\right]$$

$$\searrow l\left[\,\mathsf{c}[\,\mathsf{a}[\,0\,] \mid \mathsf{a}[\,0\,] \mid \mathsf{b}[\,S\,]\,] \,\|\, P\{T/X\} \mid P\{T'/X\}\,\right]$$

The cut operation cuts the two subtrees T and T' identified by the path expression $\mathsf{c/a}$ and spawns one copy of P for each subtree.

Now we give an example to illustrate run and the substitution of local references:

$$S = \mathsf{a}[\,\mathsf{b}[\,\Box\mathsf{go}\, m.\mathsf{go} \circlearrowleft .Q\,] \mid \mathsf{b}[\,\Box\mathsf{update}_{./../\mathsf{c}}(\chi, T).P\,]\,]$$

$$l\left[\,S \,\|\, \mathsf{run}_{\mathsf{a/b}}\,\right] \searrow l\left[\,S \,\|\, \mathsf{go}\, m.\mathsf{go}\, l.Q \mid \mathsf{update}_{\mathsf{a/b}/../\mathsf{c}}(\chi, T).P\,\right]$$

The data S is not affected by the run operation, which has the effect of spawning the two processes found by path $\mathsf{a/b}$. Note how the local path $./../\mathsf{c}$ has been resolved into the completed path $\mathsf{a/b}/../\mathsf{c}$, and $\circlearrowleft$ has been substituted by l.

3 Dynamic Web Data at Work

We give some examples of dynamic web data modelled in X$d\pi$.

Web Services. In the introduction, we described the hyperlink example. Here we generalise this example to arbitrary web services. We define a web service c with parameters $\tilde{z}$, body P, and type of result specified by the distinct variables $\tilde{w}$ bound by P:

$$\mathsf{Define}\ c(\tilde{z})\ \mathsf{as}\ P\ \mathsf{output}\ \langle\tilde{w}\rangle \triangleq !c(\tilde{z}, l, x).\, P.\, \mathsf{go}\, l.\, \overline{x}\langle\tilde{w}\rangle$$

where l and x are fixed parameters (not in $P, \tilde{w}$) which are used to specify the return location and channel. For example, process R described in the introduction can be written $\mathsf{Define}\ get(q)\ \mathsf{as}\ \mathsf{copy}_q(X)\ \mathsf{output}\ \langle X \rangle$.

We specify a service call at l to the service c at m, sending actual parameters $\tilde{v}$ and expecting in return the result specified by distinct bound variables $\tilde{w}$:

$$l{\cdot}\mathsf{Call}\ m{\cdot}c\langle\tilde{v}\rangle\ \mathsf{return}\ (\tilde{w}).Q \triangleq (\nu b)(\mathsf{go}\, m.\bar{c}\langle\tilde{v}, l, b\rangle \mid b(\tilde{w}).Q)$$

This process establishes a private session channel b, which it passes to the web service as the unique return channel. Returning to the hyperlink example, the process Q running at l can be given by

$$!load(m, q, p).\, l{\cdot}\mathsf{Call}\ m{\cdot}get\langle q\rangle\ \mathsf{return}\ (X).\mathsf{paste}_p\langle X\rangle$$

Notice that it is easy to model subscription to continuous services in our model, by simply replicating the input on the session channel:

$$l{\cdot}\mathsf{Subscribe}\ m{\cdot}c\langle\tilde{v}\rangle\ \mathsf{return}\ (\tilde{w}).P \triangleq (\nu b)(\mathsf{go}\, m.\bar{c}\langle\tilde{v}, l, b\rangle \mid !b(\tilde{w}).P)$$

Note that some web services may take as a parameter or return as a result some data containing another service call (for example, see the *intensional parameters* of [22]). In our system the choice of when to invoke such nested services is completely open, and is left to the service designer.

XLink Base. We look at a refined example of the use of linking, along the lines of XLink. Links specify both of their endpoints, and therefore can be stored in some external repository, for example

$$\texttt{XLink}[\,\texttt{To}[\,@n{:}q\,] \mid \texttt{From}[\,@l{:}p\,] \mid \texttt{Code}[\,\Box P\,]\,]$$

$$\texttt{XLinkBase}[\,\texttt{XLink}[\,...\,] \mid ... \mid \texttt{XLink}[\,...\,]\,]$$

Suppose that we want to download from an XLink server the links associated with node p in the local repository at l.

We can define a function *xload* which takes a parameter p and requests from the XLink service *xls* at m all the XLinks originating from $@l{:}p$, in order to paste them under p at location l:

$$!xload(p).l{\cdot}\mathsf{Subscribe}\ m{\cdot}xls\langle l, p\rangle\ \mathsf{return}\ (x, y, \Box\chi)$$
$$.\mathsf{paste}_p\langle\texttt{Link}[\,\texttt{To}[\,@x{:}y\,] \mid \texttt{Code}[\,\Box\chi\,]\,]\rangle$$

Service *xls* defined below is the XLink server. It takes as parameters the two components l, p making up the `From` endpoint of a link, and returns all the pairs `To`, `Code` defined in the database for $@l{:}p$.

$$\mathsf{Define}\ xls(l, p)\ \mathsf{as}\ P\ \mathsf{output}\ \langle x, y, \Box\chi\rangle$$

$$P = \mathsf{copy}_{p_1}(@x{:}y).\mathsf{copy}_{p_2}(\Box\chi)$$
$$p_1 = \texttt{XLinkBase/XLink}[\,\texttt{From}[\,@l{:}p\,]\,]\texttt{/To}$$
$$p_2 = \texttt{XLinkBase/XLink}[\,\texttt{From}[\,@l{:}p\,] \mid \texttt{To}[\,@x{:}y\,]\,]\texttt{/Code}$$

In p_1 we use the XPath syntax $\texttt{XLink}[\,\texttt{From}[\,@l{:}p\,]\,]\texttt{/To}$ to identify the node `To` which is a son of node `XLink` and a sibling of $\texttt{From}[\,@l{:}p\,]$; similarly for p_2.

Forms. Forms enhance documents with the ability to input data from a user and then send it to a server for processing. For example, assuming that the server is at location s, that the form is at path p, and that the code to process the form result is called *handler*, we have

$$\begin{aligned}\texttt{form}[\texttt{input}[\,0\,]\ &|\,\texttt{submit}[\,\Box\mathsf{copy}_{./../\texttt{input}}(X).\mathsf{go}\ s.\overline{handler}\langle X\rangle\,]\\ &|\,\texttt{reset}[\,\Box\mathsf{cut}_{./../\texttt{input}}(X)\,]]\end{aligned}$$

where $\mathsf{run}_{p/\texttt{form/submit}}$ (or $\mathsf{run}_{p/\texttt{form/reset}}$) is the event generated by clicking on the submit (or reset) button. Some user input T can be provided by a process

$$\mathsf{paste}_{p/\texttt{form/input}}\langle T\rangle$$

and on the server there will be a handler ready to deal with the received data

$$s\left[\,S\,\|\,!handler(X).P|...\,\right]$$

This example is suggestive of the usefulness of embedding processes rather than just service calls in a document: the code to handle submission may vary from form to form, and for example some input validation could be performed on the client side.

4 Behaviour of Dynamic Web Data

In the hyperlink example of the introduction, we have stated that process Q and its specification Q_s have the same intended behaviour. In this section we provide the formal analysis to justify this claim. We do this in several stages. First, we define what it means for two X$d\pi$ networks to be equivalent. Second, we indicate how to translate X$d\pi$ into another calculus, called Xπ_2, where it is easier to separate reasoning about processes from reasoning about data. Finally, we define process equivalence and argue that Q behaves like Q_s.

Network Equivalence. We apply a standard technique for reasoning about processes distributed between locations to our non-standard setting. The network contexts are

$$C ::= -\ |\ C\,|\,N\ |\ (\nu c)\ C$$

We define a *barbed congruence* between networks which is reduction-closed, closed with respect to network contexts, and which satisfies an additional *observation relation* described using *barbs*. In our case, the barbs describe the update commands, the commands which directly affect the data.

Definition 1. *A* barb *has the form $l{\cdot}\beta$, where l is a location name and $\beta \in \{\mathit{update}_p\}_{p\in\mathcal{E}}$. The* observation relation*, denoted by $N\downarrow_{l\cdot\beta}$, is a binary relation between X$d\pi$-networks and barbs defined by*

$$N\downarrow_{l\cdot\mathsf{update}_p} \iff \exists\, C[-], S, \chi, U, P, Q.\ N \equiv C[l\left[\,S\,\|\,\mathsf{update}_p(\chi,U).P\,|\,Q)\,\right]]$$

that is, N contains a location l with an update_p *command. The* weak observation relation*, denoted $N \Downarrow_{l\cdot\beta}$, is defined by*

$$N \Downarrow_{l\cdot\beta} \quad \textit{iff} \quad \exists N'.\, N \searrow N' \wedge N' \downarrow_{l\cdot\beta}$$

Observing a barb corresponds to observe at what points in some data-tree a process has the capability to read or write data.

Definition 2. *Barbed congruence ($\simeq_b$) is the largest symmetric relation $\mathcal{R}$ on $Xd\pi$-networks such that $N\,\mathcal{R}\,M$ implies*

- *N and M have the same barbs: $N \downarrow_{l\cdot\beta} \Rightarrow M \Downarrow_{l\cdot\beta}$;*
- *R is reduction-closed: $N \searrow N' \Rightarrow (\exists M'.M \searrow^* M' \;\wedge\; N'\,\mathcal{R}\; M')$;*
- *R is closed under network contexts: $\forall C.C[N]\,\mathcal{R}\,C[M]$.*

Example 2. Our first example illustrates that network equivalence does not imply that the initial data trees need to be equivalent:

$$N \triangleq l\left[\mathsf{b}[\,0\,] \,\|\, !\mathsf{paste}_{\mathsf{b}}\langle \mathsf{a}[\,0\,]\rangle \mid !\mathsf{cut}_{\mathsf{b}}(X)\,\right]$$

$$M \triangleq l\left[\mathsf{b}[\,\mathsf{a}[\,0\,] \mid \mathsf{a}[\,0\,]\,] \,\|\, !\mathsf{paste}_{\mathsf{b}}\langle \mathsf{a}[\,0\,]\rangle \mid !\mathsf{cut}_{\mathsf{b}}(X)\,\right]$$

We have $N \simeq_b M$ since each state reachable by one network is also reachable by the other, and vice versa.

An interesting example of non-equivalence is

$$l\left[T \,\|\, \mathsf{update}_p(X, X).0\right] \not\simeq_b l\left[T \,\|\, 0\right]$$

Despite this particular update (copy) command having no effect on the data and continuation, we currently regard it as observable since it has the capability to modify the data at p, even if it does not use it.

Example 3. Our definition of web service is equivalent to its specification. Consider the simple networks

$$N = l\left[T \,\|\, l\cdot\mathsf{Call}\ m\cdot c\langle\tilde{v}\rangle\ \mathsf{return}\ (\tilde{w}).Q|R\right] \;\; N_s = l\left[T \,\|\, \mathsf{go}\ m.P\{\tilde{v}/\tilde{z}\}.\mathsf{go}\ l.Q|R\right]$$

$$M = m\left[S \,\|\, \mathsf{Define}\ c(\tilde{z})\ \mathsf{as}\ P\ \mathsf{output}\ \langle\tilde{w}\rangle \mid R'\,\right]$$

If c does not appear free in R and R', then

$$(\nu c)(N \mid M) \simeq_b (\nu c)(N_s \mid M)$$

A special case of this example is the hyperlink example discussed in the introduction. The restriction c is used to prevent the context providing any competeing service on c. It is clearly not always appropriate however to make a service name private. An alternative approach is to introduce a linear type system, studied for example in [23], to ensure service uniqueness.

Separation of Data and Processes. Our aim is to define when two processes are equivalent in such a way that, when the processes are put in the same position in a network, the resulting networks are equivalent. In the technical report [7], we introduce the Xπ_2-calculus, in which the location structure is pushed locally to the data and processes, in order to be able to talk directly about processes. We translate the X$d\pi$-calculus in the Xπ_2-calculus, and equate X$d\pi$-equivalence with Xπ_2-equivalence.

Here we just summarise the translation and its results using the hyperlink example:

$$N = l\,\big[\,\texttt{Link[To[}\,@m{:}q\,\texttt{]}\,|\,\texttt{Code[}\,\Box P\,\texttt{]]}\;\|\;Q\,\big]\;|\;m\,\big[\,S\;\|\;R\,\big]$$

$$Q = !load(m,q,p).(\nu c)(\mathsf{go}\,m.\,\overline{get}\langle q,l,c\rangle \mid c(X).\mathsf{paste}_p\langle X\rangle)$$

$$R = !get(q,l,c).\mathsf{copy}_q(X).\mathsf{go}\,l.\bar{c}\langle X\rangle$$

The translation to Xπ_2 involves pushing the location structure, in this case the l and m, inside the data and processes. We use $\llparenthesis N\rrparenthesis$ to denote the translated data and $\llparenthesis S\rrparenthesis$ to denote the translation of tree S; also $\llbracket N\rrbracket$ for the translated processes and $\llbracket P\rrbracket_l$ for the translation of process P which depends on location l. Our hyperlink example becomes

$$\llparenthesis N\rrparenthesis = \{l \mapsto \texttt{Link[To[}\,@m{:}q\,\texttt{]}\,|\,\texttt{Code[}\,\Box\llbracket P\rrbracket_{\circlearrowleft}\,\texttt{]]}, m \mapsto \llparenthesis S\rrparenthesis\}$$

$$\llbracket N\rrbracket = !l{\cdot}load(m,q,p).(\nu c)(l{\cdot}\mathsf{go}\,m.\,\overline{m{\cdot}get}\langle q,l,c\rangle) \mid l{\cdot}c(X).\mathsf{paste}_p\langle X\rangle \mid$$

$$!m{\cdot}get(q,l,c).m{\cdot}\mathsf{copy}_q(X).m{\cdot}\mathsf{go}\,l.\overline{l{\cdot}c}\langle X\rangle$$

There are several points to notice. The data translation $\llparenthesis\_\rrparenthesis$ assigns locations to translated trees, which remain the same except that the scripted processes are translated using the self location $\circlearrowleft$: in our example $\Box P$ is translated to $\Box\llbracket P\rrbracket_{\circlearrowleft}$. The use of $\circlearrowleft$ is necessary since it is not pre-determined where the scripted process will run. In our hyperlink example, it runs at l. With an HTML form, for example, it is not known where a form with an embedded scripted process will be required. The process translation $\llbracket\_\rrbracket$ embeds locations in processes. In our example, it embeds location l in Q and location m in R. After a migration command, for example the $\mathsf{go}\,m.\,\_$ in Q, the location information changes to m, following the intuition that the continuation will be active at location m.

The crucial properties of the encoding are that it preserves the observation relation and is fully abstract with respect to barbed congruence, where the barbed congruence for Xπ_2 is analogous to that for X$d\pi$.

Lemma 1. *$N \downarrow_{l\cdot\beta}$ if and only if $(\llparenthesis N\rrparenthesis, \llbracket N\rrbracket) \downarrow_{l\cdot\beta}$.*

Theorem 1. *$N \simeq_b M$ if and only if $(\llparenthesis N\rrparenthesis, \llbracket N\rrbracket) \simeq_b (\llparenthesis M\rrparenthesis, \llbracket M\rrbracket)$.*

Process Equivalence. We now have the machinery to define process equivalence. We use the notation (D, P) to denote a network in Xπ_2, where D stands for located trees and P for located processes. A network (D, P) is well formed if and only if $(D, P) = ((\![N]\!), [\![N]\!])$ for some X$d\pi$ network N.

Definition 3. *Processes P and Q are* barbed equivalent*, denoted $P \sim_b Q$, if and only if, for all D such that (D, P) is well formed, $(D, P) \simeq_b (D, Q)$.*

Example 4. Recall the web service example in Example 3. The processes below are barbed equivalent:

$$Q_1 = [\![l{\cdot}\mathsf{Call}\ m{\cdot}c\langle\tilde{v}\rangle\ \mathsf{return}\ (\tilde{w}).Q]\!]_l \qquad Q_2 = [\![\mathsf{go}\ m.P\{\tilde{v}/\tilde{z}\}.\mathsf{go}\ l.Q]\!]_l$$

$$P_0 = [\![\mathsf{Define}\ c(\tilde{z})\ \mathsf{as}\ P\ \mathsf{output}\ \langle\tilde{w}\rangle]\!]_m$$

$$(\nu c)(Q_1 \,|\, P_0) \sim_b (\nu c)(Q_2 \,|\, P_0)$$

Example 5. We show that we can replicate a web service in such a way that the behaviour of the system is the same as the non-replicated case. Let internal nondeterminism be represented as $P \oplus Q \triangleq (\nu a)(\overline{a} \,|\, a.P \,|\, a.Q)$, where a does not occur free in P, Q. We define two service calls to two interchangeable services, service R_1 on channel c and R_2 on channel d:

$$Q_1 = [\![l{\cdot}\mathsf{Call}\ m{\cdot}c\langle\tilde{v}\rangle\ \mathsf{return}\ (\tilde{w}).Q]\!]_l \qquad Q_2 = [\![l{\cdot}\mathsf{Call}\ n{\cdot}d\langle\tilde{v}\rangle\ \mathsf{return}\ (\tilde{w}).Q']\!]_l$$

$$P_m = [\![\mathsf{Define}\ c(\tilde{z})\ \mathsf{as}\ P_1\ \mathsf{output}\ \langle\tilde{w}\rangle]\!]_m \qquad P_1 = [\![\mathsf{go}\ n.\overline{d}\langle\tilde{z}, l, x\rangle \oplus R_1]\!]_m$$

$$P_n = [\![\mathsf{Define}\ d(\tilde{z})\ \mathsf{as}\ P_2\ \mathsf{output}\ \langle\tilde{w}\rangle]\!]n \qquad P_2 = [\![\mathsf{go}\ m.\overline{c}\langle\tilde{z}, l, x\rangle \oplus R_2]\!]_n$$

We can show that, regardless of which service is invoked, a system built out of these processes behaves in the same way:

$$Q_1 \,|\, P_m \,|\, P_n \sim_b Q_2 \,|\, P_m \,|\, P_n$$

We can also show a result analogous to the single web-service given in Example 4. Given the specification process

$$Q_s = [\![\mathsf{go}\ m.R_1\{\tilde{v}/\tilde{z}\}.\mathsf{go}\ l.Q \oplus \mathsf{go}\ n.R_2\{\tilde{v}/\tilde{z}\}.\mathsf{go}\ l.Q']\!]_l$$

we can prove the equivalence

$$(\nu c, d)(Q_1 \,|\, P_m \,|\, P_n) \sim_b (\nu c, d)(Q_s \,|\, P_m \,|\, P_n)$$

where the restriction of c and d avoids competing services on the same channel. In particular, if $Q = Q'$, $R_1 = R_2$, and R_1 does not have any barb at m, then

$$(\nu c, d)(Q_1 \,|\, P_m \,|\, P_n) \ \sim_b \ (\nu c, d)([\![\mathsf{go}\ m.R_1\{\tilde{v}/\tilde{z}\}.\mathsf{go}\ l.Q]\!]_l \,|\, P_m \,|\, P_n).$$

This equivalence illustrates that we can replicate a web service without a client's knowledge.

These equivalences are known to be difficult to prove. In [7], we provide a method for proving such equivalences, by mimicking our update commands using π-calculus interaction and using the standard bisimulation techniques of the π-calculus. In particular, we develop a bisimulation equivalence $\approx$ with the property that, given two X$a\pi_2$ processes P, Q, then $P \approx Q$ implies $P \sim_b Q$. Here we just indicate the proof technique by informally explaining why

$$Q_1 \,|\, P_m \,|\, P_n \approx Q_2 \,|\, P_m \,|\, P_n.$$

We show that process Q_2 can simulate Q_1. Process Q_1 starts by sending an output process to m, ready to invoke service c. Process Q_2 can send an output process to n, ready to invoke service d. If the internal non-determinism selects the first branch then, after another step of migration from n to m, we have reached a state identical to the current state of Q_1. In both cases, at l there is a process waiting on a restricted channel, and at m the same output process ready to trigger service c. The proof that Q_1 can simulate Q_2 is symmetric. There is much hidden complexity in this example.

5 Concluding Remarks

This paper introduces X$d\pi$, a simple calculus for describing the interaction between data and processes across distributed locations. We use a simple data model consisting of unordered labelled trees, with embedded processes and pointers to other parts of the network, and π-processes extended with an explicit migration primitive and an update command for interacting with data. Unlike the π-calculus and its extensions, where typically data is encoded as processes, the X$d\pi$-calculus models data and processes at the same level of abstraction, enabling us to study how properties of data can be affected by process interaction.

Alex Ahern has developed a prototype implementation, adapting the ideas presented here to XML standards. The implementation embeds processes in XML documents and uses XPath as a query language. Communication between peers is provided through SOAP-based web services and the working space of each location is endowed with a process scheduler based on ideas from PICT [24]. We aim to continue this implementation work, perhaps incorporating ideas from other recent work on languages based on the π-calculus [25, 26].

Active XML [8] is probably the closest system to our X$d\pi$-calculus. It is based on web services and service calls embedded in data, rather than π-processes. There is however a key difference in approach: Active XML focusses on modelling data transformation and delegates the role of distributed process interaction to the implementation; in contrast, process interaction is fundamental to our model. There are in fact many similarities between our model and features of the Active XML implementation, and we are in the process of doing an in-depth comparison between the two projects.

Developing process equivalences for X$d\pi$ is non-trivial. We have defined a notion of barbed equivalence between processes, based on the update operations that processes can perform on data, and have briefly described a proof method for proving such equivalences between processes. There are other possible definitions of observational equivalence, and a comprehensive study of these choices will be essential in future. We also plan to adapt type theories and reasoning techniques studied for distributed process calculi to analyse properties such as boundary access control, message and host failure, and data integrity. This paper has provided a first step towards the adaptation of techniques associated with process calculi to reason about the dynamic evolution of data on the Web.

Acknowledgements

We would like to thank Serge Abiteboul, Tova Milo, Val Tannen, Luca Cardelli, Giorgio Ghelli and Nobuko Yoshida for many stimulating discussions.

References

1. Abiteboul, S., Bonifati, A., Cobena, G., Manolescu, I., Milo, T.: Dynamic XML documents with distribution and replication. In: Proceedings of ACM SIGMOD Conference. (2003)
2. Sahuguet, A., Tannen, V.: Resource Sharing Through Query Process Migration. University of Pennsylvania Technical Report MS-CIS-01-10 (2001)
3. Milner, R., Parrow, J., Walker, J.: A calculus of mobile processes, I and II. Information and Computation **100** (1992) 1–40,41–77
4. Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press (2001)
5. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: Proceedings of ECOOP. Volume 512 of LNCS., Berlin, Heidelberg, New York, Tokyo, Springer-Verlag (1991) 133–147
6. Carbone, M., Maffeis, S.: On the expressive power of polyadic synchronisation in π-calculus. Nordic Journal of Computing **10** (2003) 70–98
7. Gardner, P., Maffeis, S.: Modeling dynamic Web data. Imperial College London Technical Report (2003)
8. Abiteboul, S. et al.: Active XML primer. INRIA Futurs, GEMO Report number 275 (2003)
9. Sahuguet, A.: ubQL: A Distributed Query Language to Program Distributed Query Systems. PhD thesis, University of Pennsylvania (2002)
10. Kemper, A., Wiesner, C.: Hyperqueries: Dynamic distributed query processing on the internet. In: Proceedings of VLDB'01. (2001) 551–560
11. Braumandl, R., Keidl, M., Kemper, A., Kossmann, D., Kreutz, A., Seltzsam, S., Stocker, K.: Objectglobe: Ubiquitous query processing on the internet. To appear in the VLDB Journal: Special Issue on E-Services (2002)
12. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: from relations to semistructured data and XML. Morgan Kaufmann (2000)
13. Cardelli, L., Ghelli, G.: A query language based on the ambient logic. In: Proceedings of ESOP'01). Volume 2028 of LNCS., Springer (2001) 1–22
14. Honda, K., Yoshida, N.: On reduction-based process semantics. Theoretical Computer Science **151** (1995) 437–486
15. Hennessy, M., Riely, J.: Resource access control in systems of mobile agents. In: Proceedings of HLCL '98. Volume 16.3 of ENTCS., Elsevier (1998) 3–17

16. Gordon, A., Pucella, R.: Validating a web service security abstraction by typing. In: Proceedings of the 2002 ACM Workshop on XML Security. (2002) 18–29
17. Godskesen, J., Hildebrandt, T., Sassone, V.: A calculus of mobile resources. In: Proceedings of CONCUR'02. LNCS (2002)
18. Sahuguet, A., Pierce, B., Tannen, V.: Distributed Query Optimization: Can Mobile Agents Help? (Unpublished draft)
19. Bierman, G., Sewell, P.: Iota: a concurrent XML scripting language with application to Home Area Networks. University of Cambridge Technical Report UCAM-CL-TR-557 (2003)
20. World Wide Web Consortium: XML Path Language (XPath) Version 1.0. (available at http://w3.org/TR/xpath)
21. Berger, M.: Towards Abstractions for Distributed Systems. PhD thesis, Imperial College London (2002)
22. Abiteboul, S., Benjelloun, O., Milo, T., Manolescu, I., Weber, R.: Active XML: A data-centric perspective on Web services. Verso Report number 213 (2002)
23. Berger, M., Honda, K., Yoshida, N.: Linearity and bisimulation. In: Proceedings of FoSSaCS'02, LNCS (2002) 290–301
24. Pierce, B.C., Turner, D.N.: Pict: A programming language based on the pi-calculus. In: Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press (2000)
25. Conchon, S., Fessant, F.L.: Jocaml: Mobile agents for Objective-Caml. In: Proceedings of ASA'99/MA'99, Palm Springs, CA, USA (1999)
26. Gardner, P., Laneve, C., Wischik, L.: Linear forwarders. In: Proceedings of CONCUR 2003. Volume 2761 of LNCS., Springer (2003) 415–430

Semantics of Objectified XML Constraints*

Suad Alagić and David Briggs

Department of Computer Science
University of Southern Maine
Portland, ME 04104-9300
{alagic,briggs}@cs.usm.maine.edu

Abstract. The core of a model theory for a functional object-oriented data model extended with XML-like types is presented. The object-oriented component of this integrated paradigm is based on Featherweight Java and XML is represented by regular expression types. The main contributions are in extending both with general logic-based constraints and establishing results on schema and database evolution by inheritance that respects database integrity requirements. The paper shows that formally defined semantics of this integrated paradigm does indeed exist and in fact may be constructed in a model-theoretic fashion. The generality of the developed model theory and its relative independence of a particular logic basis makes it applicable to a variety of approaches to XML (as well as object-oriented) constraints. A pleasing property of this model theory is that it offers specific requirements for semantically acceptable evolution of these sophisticated schemas and their databases.

1 Introduction

Several industrial tools and very few research results have been developed that address the problems of integrated environments that include both a major object-oriented programming language (such as Java) and a typed XML model (such as XML Schema [20]). The pragmatic importance of these efforts is based on complex applications in which data in XML format is processed in a general computational manner.

In order to guarantee correctness in those situations the rules of particular type systems (object-oriented and XML) must be respected in the first place. This is where the first problem is, as the two type systems, in spite of some similarities, do not match. One research challenge is to develop an integrated type system as a basis of an integrated environment. But even if an integrated type system is available there still remains a fundamental issue whether the semantics of the data is respected, i.e., whether the integrity constraints associated with an XML schema are enforced when the data is processed in an object-oriented

* This material is based upon work supported in part by NSF under grant number IIS-9811452 and in part by the Institute for Information Sciences of the University of Southern Maine.

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 147–165, 2004.

language. There is no way of enforcing this requirement in a language like Java other than doing it procedurally, and this is true for most object-oriented languages. Most object oriented languages (Eiffel is an exception) lack declarative, logic-based constraint specification capabilities and the underlying technology to enforce them. Industrial tools such as JDOM [15] or JAXB [14] suffer from the same limitations.

A schema in this integrated environment would in general be equipped with both XML constraints associated with document types and object-oriented constraints. In general, both should be expressed in a constraint language based on a suitable logic. In this paper we take up the challenge of developing a model theory for such schemas. This model theory has been developed with a very specific pragmatic problem in mind: maintaining the integrity constraints when these sophisticated schemas and the associated data evolve. The model-theoretic requirements developed in the paper guarantee this fundamental property.

The object-oriented model and its type system in this paper are based on Featherweight Java (FJ) [13], a functional simplification of Java. A formally defined XML-like model used in this paper is based on regular expression types as presented in [12]. Both of these models are typed, but their type systems are quite different, which is the first problem to be addressed. However, a more ambitious goal is extending both paradigms with typed, logic-based constraints absent from either one.

FX [11] presents one way of integrating the two type systems. We take a different approach that is more suitable for database schemas. Just like FJ or FX, our paradigm remains functional, but it still fits the problem well as it allows typical database integrity constraints such as keys and inclusion dependencies to be expressed. Likewise, although regular expression types do not include all the generality of typed XML models, they do cover the core, and allow a formal definition that is a basis for a model theory. In this paper we show that a model theory of the above described integrated paradigm does indeed exist and that it leads naturally to requirements on semantically acceptable evolution of these sophisticated schemas.

2 Document Values and Document Objects

A major distinctive feature of the integration presented in this paper is that XML documents may be viewed both as complex structured values as well as wrapped into objects which are instances of a class `Document`. When XML-like documents are viewed as structured values, they are not equipped with object-oriented identity and their complex structure is determined by the rules for regular expression types as illustrated below. The position of document types, whose instances are structured values, in the integrated type system proposed in this paper, is similar to the position of simple types such as `int` in Java. The difference is that document types are not predefined and they are not simple either. This is in accordance with other data models that allow structured values that are not necessarily objects, such as for example ODMG [9] or O2. This is one significant departure from the type system of FX [11].

A document type is in general augmented with constraints. There are many possibilities for the constraint language, but the approach we propose accommodates the commonly used ones. In all the cases that we consider in this paper constraints are sentences with all variables universally quantified. For illustrative purposes in this paper we use constraints expressible as Horn clauses with equality.

Signatures for document types include type names of simple or document types, signatures for tree types, concatenation, union and iteration of document types. Letting T range over document types, an abstract syntax for them includes the following.

- A named document type N.
- Tree type signature $< l > [T]$ or $< l > [P]$, with $l \in L$ where L stands for a set of labels and P ranges over simple types.
- Empty document type ( ).
- Concatenation T_1, T_2 of document types T_1 and T_2.
- Union $T_1 \mid T_2$ of document types T_1 and T_2.
- Iteration T^* of a document type T.

An example of a regular expression document type in a more concrete syntax is given below:

```
public document qualification [
     <Year>[String],
     (<Degree>[<School>[String],
               <AbbDegree>[String]]
             |
      <Certificate>[String])]
```

In this example a `qualification` instance consists of a `Year` in which it was received, and is either a `Degree`, in which case it has a granting `School` and an `AbbDegree` characterization, or is a `Certificate`.

Constraints for a document type are illustrated in the example below. Any logic language for constraints involving regular document types would require operators appropriate for these structured types, such as a binary *ischild* and list operations. We do not provide a complete list of these here, but do use one unnamed ternary predicate relating documents, sequences of labels, and tree documents: *doc seq tr* is true when *tr* is a subtree within *doc* and the sequence of node labels to *tr* is given by *seq*.

Quantified variables have their types specified with the quantifier and range over the values of their type. We assume conventional operators for manipulating simple types and accessing and constructing document type instances. A distinctive feature of the approach is that it is largely independent of a specific underlying logic and thus applies to different constraint languages.

```
forAll qualification Q,
       <Year>[String] Y,
       <School>[String] S,
       <AbbDegree>[String] D,
       <Certificate>[String] C
```

```
   ((D = "PhD" <- S = "UMASS",
                  Y = "2003",
                  Q<Year>Y,
                  Q<Degree><School>S,
                  Q<Degree><AbbDegree>D),
    (C = "" <- Y = "",
               Q <Year>Y,
               Q<Certificate>C))
```

The first clause asserts that if a `qualification` instance is a `Degree` awarded in 2003 from UMASS, then it must be a PhD. The second clause asserts that if a qualification is a `Certificate` and the `Year` is the empty string, then the `Certificate` is the empty string as well.

In an expression such as `Q<Degree><School> S`, by the typing of the quantified variables `Q` refers to a fixed but arbitrary qualification document type instance, `S` refers to an instance of `<School>[String]`, and `Q<Degree><School> S` indicates that `S` is a subtree within `Q` such that the path from `Q` to `S` is labeled by `<Degree><School>`. In general, expressions of the form $e < l_1 >< l_2 > \cdots < l_n > t$ are meaningful whether the first argument is a tree or a list of trees, because the label of the root of e is not included in the path. The last label of the sequence of labels, however, is the label of the root of the subtree t. Additional operators appropriate for extracting a node's list of children and manipulating that list would be included in a full version of the language.

The structural aspect of such a constrained type definition determines a set of values in a conventional way, and the constraints then serve as a filter whereby values that do not satisfy the constraints are discarded as invalid instances of that type.

In order to view a document type as an object type, a document object type must be derived by inheritance from the abstract class `Document`. This abstract class contains methods that operate on complex document structure along with the associated constraints. The abstract class `Document` thus plays a role that is similar to the wrapper class `Number` in Java. The difference is that document values have complex structure and document object types may naturally have subtypes, unlike Java wrapper classes such as `Integer`.

The example below illustrates the constraints for the method `equals` of the root class `Object`. These constraints assert that the method `equals` of document object types is defined as an equivalence relation, a condition impossible to specify or enforce in a reasonable manner in Java or FJ because of lack of any constraint capabilities.

A specific document object type (`Qualification` in the example below) extends the class `Document`. It necessarily contains a constructor whose argument is a complex value of the corresponding regular document type (`qualification` in the example below).

```
abstract class Document
{ ...
 forAll Document  D,D1,D2,D3
 ((D.equals(D)),
  (D2.equals(D1) <- D1.equals(D2)),
  (D1.equals(D3) <- D1.equals(D2), D2.equals(D3)))}

class Qualification extends Document
{... Qualification(qualification q);
     qualification valueOf();}

Collection<Qualification> docs;
```

In the FX integration all regular expression values are instances of a special final class Seq, and an instance of any class may be used as a label of a tree node. The integrated type system permits a liberal intermingling of the constructors from each paradigm. Our approach is less symmetric than the FX approach in that XML features are added to the object oriented paradigm, but the regular document type system remains intact. Document types can be used as data fields in classes and may appear in method signatures, but object types cannot be used within regular document types, either as labels or as leaf values.

As noted above, our approach allows instances of regular expression types to be viewed as structured values or wrapped as objects. This is an alternative to FX where regular object types do not have the basic features such as methods or fields and do not participate properly in the inheritance hierarchy. Hence, they are not really object types at all. This is why we propose the approach presented in this paper as a better alternative to the integration of XML and the object-oriented paradigm. The advantages of our alternative are particularly important for database models in which complex structured values as well as complex objects are quite common.

3 Class Signatures

Specification of an application model in this approach consists of two components. The first component is the structural description and the second specifies additional semantic properties via constraints. In this section we discuss the structural aspects that are formalized by class signatures.

The signature Σ_C of a class C specifies the structural and operational features of a class and consists of the following components:

- $Sorts(C)$ - a set that includes at least types appearing in inner types, fields and method signatures of C.
- $Fields(C)$ - a sequence $F_1\ f_1; F_2\ f_2;\ \ldots; F_m\ f_m$ of field signatures of C.
- $Methods(C)$ - a set of method signatures of C. A signature for a method m has the form $C_A\ m(C_{A_1}\ x_1, C_{A_2}\ x_2, \ldots, C_{A_n}\ x_n)$

Class signatures are equipped with the subtyping partial order <:, based on the subtyping order of sorts, and defined as follows:

$\Sigma_B <: \Sigma_A$ iff `B extends A` or `B implements A`. In that case we have:

- $Sorts(A) \subseteq Sorts(B)$ and the orderings of $Sorts(A)$ and $Sorts(B)$ agree.
- $Fields(B) = Fields(A); F_{B_{m+1}}; F_{B_{m+2}}; \ldots; F_{B_n}$
- Let $C_A\ m(C_{A_1}\ x_1, C_{A_2}\ x_2, \ldots, C_{A_n}\ x_n)$ be a signature of a method of A ($m \in Methods(A)$), then $m \in Methods(B)$ and the signature of the method m of B, $C_B\ m(C_{B_1}\ x_1, C_{B_2}\ x_2, \ldots, C_{B_n}\ x_n)$ must satisfy the conditions:
  - $C_{A_i} <: C_{B_i}$ and $C_{B_i} <: C_{A_i}$ (type equivalence) for $i = 1, \ldots, n$ and
  - $C_B <: C_A$
- If $T_2 <: T_1$ holds for regular document types, and D_2 and D_1 are their corresponding wrapper classes (object document types), then $D_2 <: D_1$ and hence the class D_2 extends the class D_1 as in the Java type system.

Note that the subtyping order for object types is a partial order and it is a preorder for regular document types to which structural type equivalence applies. Unlike the Java subtyping by definition, subtyping for document types is structural [11]. This presents one challenge for producing an integrated type system. The rules for structural subtyping for regular document types (not presented here) are much more complex [12] than the Java rules, but the nature of the subtyping is actually very intuitive and model-oriented: $T_2 <: T_1$ iff the set of values of T_2 is a subset of the set of values of T_1.

4 Database Schemas

A major extension with respect to FJ is that specification of a class (or a document type) consists of two components, the signature and the constraints. The constraints associated with a class or a document type are specified as sentences, i.e., formulas in a suitable logic-based constraint language with all the variables quantified. The notation for such a specification for a class A is (Σ_A, E_A) where Σ_A denotes a class signature for A and E_A denotes a set of sentences based on Σ_A and the chosen logic. This in particular applies to database schemas which have some additional specific properties.

A database schema is a pair (Σ_D, E_D) *where*

- *D is a class such that* `D extends Database` *(directly or transitively)*
- *Σ_D is the class signature for D.*
- *E_D is a set of sentences (integrity constraints)*

A distinguished predefined class `Database` captures the basic features of database schemas. It represents a functional version of the same class in the ODMG model [9] with an important difference: it is equipped with logic-based constraints.

```
public class Database
{public Database(String name);
 public final boolean open();
 public final boolean close();
 public final Object lookUp(String name);
 public final Database bind(Object x,String name);
forAll Database D, Object x, String n
((D.bind(x,n).lookup(n).equals(x)),
 ...)
}
```

The method bind of this class binds a name (the second argument) to an object of any type (the first argument of type Object). As Class extends Object, a database contains bindings for both classes and objects. The method lookUp returns an object of a database bound to a name. Note that the only constraint given in the above class specifies the semantic relationship between the methods bind and lookUp. The class Database contains additional constraints, as do the specific application-oriented classes derived from it.

An example of a database schema is given below. It includes a document type qualification (which is not an object type), object types Employee and EmployeeCollection, an instance variable dbEmployees which is a database collection, methods such as hireEmployee and fireEmployee, a key constraint and a method-related constraint.

```
public class Personnel extends Database {
 public document qualification [
  ...  // as defined above ]
 public interface Employee{
  String name();
  String ssn();
  Float salary();
  qualification* qualifications();
}
 public class EmployeeCollection
             implements Collection<Employee>{...}
 protected EmployeeCollection dbEmployees;
 protected boolean hireEmployee(Employee e);
 protected boolean fireEmployee(Employee e);

 forAll Employee E,E1,E2
 ((dbEmployees.contains(E) <- hireEmployee(E)),
  (E1.equals(E2) <- dbEmployees.contains(E1),
                    dbEmployees.contains(E2),
                    E1.ssn().equals(E2.ssn())))
}
```

5 Models

The first step toward a model theory requires definition of models for class signatures only (i.e., without constraints). A model for a class signature interprets a type signature as a family of sets and method signatures as functions (FJ is a functional Java language). A model $\mathcal{M}_A$ for a class signature Σ_A consists of the following:

- A collection of sets $\{M_C\}$, one set M_C for every type C in $Sorts(A)$. If D and C are in $Sorts(A)$ then $D <: C$ implies $M_D \subseteq M_C$.
- A particular model constructed in accordance with the FX approach [11] is defined as follows:
  If C is an object type (a class or an interface)
  $M_C = \{new\ D(a_1, a_2, \ldots, a_n) \mid D\ <:\ C\ and\ a_i \in M_{D_{F_i}}\}$ where D_{F_i} is the type of the ith field of a class D.
  Although we base most of the paper on this particular model, the developed model theory allows a more general treatment.
- For each method m in Σ_A with the signature $C\ m(C_1\ x_1, C_2\ x_2, \ldots, C_n\ x_n)$ a function $f_m : M_w \rightarrow M_C$ where $M_w = M_A \times M_{C_1} \times \ldots \times M_{C_n}$ with $w = AC_1 \ldots C_n$.

As in Java and FJ, the partial order of classes has a maximum value, the class `Object`, and the definition of the model for `Object` is the starting point for the definition of models for all classes. A model of any other class is defined with respect to the model of its superclass by extending that model as defined above.

For a document type D the set M_D is determined according to the following rules where t_i refers to a tree instance and s_i refers to a sequence of trees:

- If D is of the form $< l > [T]$ then
  $M_D = \{< l > [t_1, t_2, \ldots, t_n] \mid (t_1, t_2, \ldots, t_n) \in M_T\}$
- If D is of the form $< l > [P]$ then
  $M_D = \{< l > [v] \mid v \in M_P\}$, where M_P is the interpretation of the simple type P
- If D is the empty document type, then $M_D = \{[]\}$
- If D is a concatenation T_1, T_2 then $M_D = \{s_1 s_2 \mid s_1 \in M_{T_1}, s_2 \in M_{T_2}\}$
- If D is a union $T_1 \mid T_2$ then $M_D = M_{T_1}\ \cup\ M_{T_2}$
- If D is the iteration T^* then
  $M_D = \{s_1 s_2 \cdots s_n \mid n >= 0, for\ all\ k = 1, 2, \ldots n,\ s_k \in M_T\}$

The above definitions lead naturally to the following definition of a default value of a given type:

- $default_{Object} = new\ Object()$
- If T is a simple type, $default_T$ is its default value as defined for Java.
- $default_T = []$ where T is a regular document (value) type and [] is the empty document.
- $default_C = new\ C(default_{F_1}, default_{F_2}, \ldots, default_{F_n})$ where C is a class with a sequence of fields $F_1\ f_1; F_2\ f_2; \ldots, F_n\ f_n$ and $default_{F_i}$ is the default value of type F_i.

The subtyping relation is semantic in nature, i.e., it is defined in terms of the underlying models. It has particularly desirable properties related to the substitution (abstraction) function that allows an object of a subtype to be viewed as an object of its supertype. If Σ_C is a signature for a type C, let $Mod(\Sigma_C)$ denote a small set containing all models of Σ_C.

The availability of default values makes it possible to prove the following result:

Proposition 1
If Σ_B <: Σ_A then there exists an abstraction function
$absF : Mod(\Sigma_B) \to Mod(\Sigma_A)$.

Proof. If $\Sigma_B <: \Sigma_A$ then $Sorts(A) \subseteq Sorts(B)$. Let $\mathcal{M}_B$ be a model for Σ_B so that we have $\mathcal{M}_B = \{M_C \mid C \in Sorts(B)\}$. From $\mathcal{M}_B$ we obtain a Σ_A model $absF(\mathcal{M}_B)$ as a family $\{M_C \mid \ C \in Sorts(A)\}$.

Let $C_B\ m(C_{B_1}\ x_1, C_{B_2}\ x_2, \ldots, C_{B_n}\ x_n)$ be a signature of a method of B and $C_A\ m(C_{A_1}\ x_1, C_{A_2}\ x_2, \ldots, C_{A_n}\ x_n)$ the signature of that method in A. From the function $f_m^B : M_B \times M_{C_{B_1}} \times \ldots \times M_{C_{B_n}} \to M_{C_B}$ of $\mathcal{M}_B$ we construct a function $f_m^A : M_A \times M_{C_{A_1}} \times \ldots \times M_{C_{A_n}} \to M_{C_A}$ of $absF(\mathcal{M}_B)$ as follows:

- We have a retraction function $retF : M_A \to M_B$ defined as
  $retF(new\ A(a_1, a_2, \ldots, a_m)) =$
  $new\ B(a_1, a_2, \ldots, a_m, default_{F_{B_{m+1}}}, \ldots, default_{F_{B_n}})$ where
  $Fields(B) = Fields(A); F_{B_{m+1}}\ f_{m+1}; \ldots; F_{B_{m+1}}\ f_{m+1}$.
- $M_{C_{B_i}} = M_{C_{A_i}}$ because $C_{B_i} <: C_{A_i}$ and $C_{A_i} <: C_{B_i}$.
- $M_{C_B} \subseteq M_{C_A}$ because $C_B <: C_A$ so we have the injection $inc_{BA} : M_{C_B} \to M_{C_A}$.
- The desired function f_m^A is constructed as $M_A \times M_{C_{A_1}} \times \ldots \times M_{C_{A_n}} \to M_B \times M_{C_{B_1}} \times \ldots \times M_{C_{B_n}} \to M_{C_B} \to M_{C_A}$, i.e.
  $f_m^A = inc_{BA} f_m^B(retF \times id_{M_{C_{B_1}}} \times \ldots \times id_{M_{C_{B_n}}})$

The subtyping rule for function types is based on the reasoning about valid composition of functions which is precisely the basis of the above existence proof. An alternative approach to constructing models for (sub)classes with satisfaction of constraints and the properties of the abstraction function are discussed in section 8.

6 Constraint Language

A constraint language for this integrated paradigm consists of sentences. The syntax of sentences is determined by the underlying type signature and the rules of the chosen logic. The type signature determines the terms of the constraint language. The definition of formulas is then based on such terms. It starts with the definition of atomic predicates and the rules of their composition into more complex formulas using the logical connectives of the underlying logic. As for atomic predicates, we require that at least suitably defined equality predicates are available both for document values and document objects.

The terms of object-oriented types and document types are defined below:

- A variable X of type T is a term of type T.
- $new\ C(e_1, e_2, \ldots, e_n)$ is an object term of type C iff (i) e_i is a term of type F_i for $i = 1, \ldots, n$, and (ii) C has a sequence of fields $F_1\ f_1; F_2\ f_2; \ldots; F_n\ f_n$.
- $e.f$ (field access) is a term of type F if (i) e is a term of object type C and (ii) C has a field f of type F.
- $e.m(e_1, e_2, \ldots, e_n)$ is a term of type A iff (i) e is a term of object type C (ii) e_i is a term of type A_i for $i = 1, \ldots, n$ (iii) C has a method m with the signature $A\ m(A_1\ a_1, A_2\ a_2, \ldots, A_n\ a_n)$
- $< l > [t_1, t_2, \ldots, t_n]$ is a tree term of a document type iff each t_i is a term of a tree document type and l is a label.
- $< l > [e]$ is a tree term of a document type if e is a term of a simple type and l is a label.
- $[e_1, e_2, \ldots, e_n]$ is a term of a sequence type iff e_i is a term of a document type for $i = 1, \ldots, n$.

Other forms of expressions will be available with the introduction of operations on document types (trees and sequences).

Regular document types, just like simple types, are equipped with standard equality denoted = (rather than == as in Java). Its definition follows the structural pattern of complex document values.

- Tree values: $< l > [s] =< l' > [s']$ iff $l = l'$ and $s = s'$.
- Sequences: $[t_1, t_2, \ldots, t_m] = [t'_1, t'_2, \ldots, t'_n]$ iff $m = n$ and $t_i = t'_i$ for $i = 1, 2, \ldots, n$.

Object types are equipped with the boolean method `equals` of the root class `Object`. The definition of this method given below reflects the definition of models for FJ object types.

- $new\ C(e_1, e_2, \ldots, e_n).equals(new\ C'(e'_1, e'_2, \ldots, e'_n))$ iff (i) $C = C'$ and (ii) $e_i = e'_i$ for $i = 1, 2, \ldots, n$ if the type of e_i and e'_i is simple or a document type, otherwise (iii) $e_i.equals(e'_i)$ for $i = 1, 2, \ldots, n$.

Atoms of the constraint language include at least the following:

- $e_1.equals(e_2)$ is an atom where e_1 and e_2 are object terms.
- $e_1.p(e_2, \ldots, e_n)$ is an atom where e_1 is an object term and p is a boolean method.
- $e_1 = e_2$ is an atom where e_1 and e_2 are terms of simple types or terms of document types.
- Predicates defined for document types (trees and sequences).

Constraints are sentences, i.e., formulas with all variables quantified. The definition of constraints given below is given for a specific, Horn clause logic with equality. For other, more expressive logics the definition of constraints would be more complex. Otherwise, the core of the model theory does not depend upon a specific logic.

- A clause has the form $(p \leftarrow p_1, p_2, \ldots, p_n)$ where $p, p_1, p_2, \ldots, p_n$ are atoms.
- A constraint is a sentence of the form
  $forAll\ T_1\ X_1, T_2\ X_2, \ldots, T_m\ X_m\ \ (clause_1, clause_2, \ldots, clause_n)$
  where T_i is a type name and X_i is a variable of type T_i for $i = 1, \ldots, m$.

7 Satisfaction

Our specification proposal generalizes over the particulars of the logic, but uses fundamental notions of signature, model, sentence, and satisfaction. We have discussed signatures and models above. Sentences express the constraints and depend upon the particular logic. Satisfaction is a relation between the set of models of a signature and its set of sentences within the logic. Its definition depends upon the logic, and we explain it for the Horn clause logic in this paper.

A model satisfies a sentence when the sentence evaluates to true for the model. A single clause of Horn clause logic has all variables universally quantified, so the sentence evaluates to true when the clause evaluates to true for every substitution of values for variables. As we are in a typed context, the substitutions must bind values of the appropriate types to the variables. Once the variables are bound to values, evaluation can proceed recursively on the structure of the terms. The details of the evaluation function $eval_\theta$ for a specific type-respecting substitution of values for variables θ follow.

- A substitution of variables θ is a family of functions $\{\theta_C : \mathcal{X}_C \rightarrow M_C\}$ where $\mathcal{X}_C$ denotes a set of variables of type C.
- The term evaluation function $eval_\theta$ is a family of functions $\{eval_\theta^C\}$ where $eval_\theta^C : Terms_C \rightarrow M_C$ and $Terms_C$ denotes the set of terms of type C.

The term evaluation function is defined as follows:

- $eval_\theta(new\ C(e_1, e_2, \ldots, e_n)) = new\ C(a_1, a_2, \ldots, a_n)$ iff
  $eval_\theta(e_i) = a_i$ for $i = 1, 2, \ldots, n$.
- $eval_\theta(e.f_k) = a_k$ iff $f_k \in fields(C)$ and $eval_\theta(e) = new\ C(a_1, a_2, \ldots, a_n)$
- $eval_\theta(e.m(e_1, e_2, \ldots, e_n)) = a$ iff
  - $eval_\theta(e) = o$
  - $f_m(o, eval_\theta(e_1), eval_\theta(e_2) \ldots, eval_\theta(e_n)) = a$
- $eval_\theta([e_1, e_2, \ldots, e_n]) = [a_1 a_2 \cdots a_n]$ iff $eval_\theta(e_i) = a_i$ for $i = 1, 2, \ldots, n$.
- $eval_\theta(< l > [t_1, t_2, \ldots, t_n]) = < l > [a_1, a_2, \ldots, a_n]$
  iff $eval_\theta(t_i) = a_i$

The definition of the constraint evaluation function $eval : Constraints \rightarrow \{false, true\}$ is, of course, based on the definition of the term evaluation function. In addition, constraints are sentences so that the constraint evaluation function is not parameterized with a substitution of variables θ. The constraint evaluation function is determined by the underlying logic. It is illustrated below for Horn clause logic.

- $eval(p \leftarrow p_1, p_2, \ldots, p_n) = true$ iff $eval_\theta(p_i) = true$ for all $i = 1, 2, \ldots, n$ implies $eval_\theta(p) = true$ for any substitution of variables θ which respects the typing constraints.
- $eval(forAll\ T_1\ X_1, T_2\ X_2, \ldots, T_m\ X_m$ $(clause_1, clause_2, \ldots, clause_n)) = true$ iff $eval(clause_i) = true$ for all $i = 1, 2, \ldots, n$

A class signature Σ augmented with a set of constraints E as defined above becomes a *class theory* (Σ, E). A model for a theory (Σ, E) must provide an interpretation of the class signature Σ in such a way that the set of constraints E is satisfied.

$\mathcal{M}$ is a model for a class theory (Σ, E) denoted $\mathcal{M} \models (\Sigma, E)$ (or just $\mathcal{M} \models E$ if Σ is understood) iff

- *$\mathcal{M}$ is a model for the class signature Σ.*
- *$eval(e) = true$ for every sentence $e \in E$ where the evaluation function eval is defined above.*

Now that we have the notion of satisfaction defined we can prove the following fundamental property of the model construction from Proposition 1.

Proposition 2

- *Let $\Sigma_B <: \Sigma_A$ and $\mathcal{M}_B$ be a model of Σ_B, with a retraction function ret_F as defined in Proposition 1.*
- *Suppose the following holds for the interpretations of all Σ_A methods m in $\mathcal{M}_B$:*
  $f^A_m = inc_{BA} f^B_m (ret_F \times \quad {}_i\, id_{M_{C_{B_i}}})$.

Then for all Σ_A sentences e, $\mathcal{M}_B \models e \Leftrightarrow absF(\mathcal{M}_B) \models e$.

Proof. The key is to show that for all terms t involving only sorts and operations from $Sorts(A)$, and for all θ, typed variable assignments sending variables of sort C to values in M_C of $\mathcal{M}_B$, $eval_\theta(t)$ in the model $\mathcal{M}_B$ is the same value as $eval_\theta(t)$ in the constructed model, $absF(\mathcal{M}_B)$. The condition on the method interpretations in $\mathcal{M}_B$ is exactly what is needed to guarantee that the method redefinitions of $absF(\mathcal{M}_B)$ do not change the way in which method invocations evaluate, so any term will evaluate to the same value in either model. Since the collection of sets is the same in both models, the quantifiers range over the same sets of values, so a formula e will evaluate to the same result in both models.

Note that if B inherits a method m without redefining it, then the condition $f^A_m = inc_{BA} f^B_m (ret_F \times \quad {}_i\, id_{C_{B_i}})$ will automatically hold.

Corollary 1
If $\Sigma_B <: \Sigma_A$, (Σ_B, E_B) and (Σ_A, E_A) are theories, and $E_A \subseteq E_B$, then for all models of Σ_B, $\mathcal{M}_B$, satisfying the conditions of Propositions 1 and 2,
$\mathcal{M}_B \models E_B \Rightarrow absF(\mathcal{M}_B) \models E_A$.

8 Consistent Schema Extensions

The availability of constraints in this integrated paradigm makes it possible to address the semantic and database integrity issues that cannot be addressed in other related paradigms (such as Java, FJ, JAXB, etc.) that are not equipped with constraints. Particularly important are the issues of semantically correct inheritance that ensures behavioral compatibility of a subclass with respect to its superclass. A related database issue is semantically acceptable schema evolution in which the extended schema conforms to the integrity constraints of its super schema.

Object-oriented schemas evolve naturally by inheritance (or extension in the Java terminology). The rules imposed by a type system guarantee type safety of such extensions. However, the type system alone, which essentially characterizes structural features, cannot deal with issues of integrity constraints. This in particular applies to Java, FJ and XML-like paradigms such as FX. A constraint-based paradigm would ensure that an extended schema conforms to the super schema with respect to database integrity. This in particular ensures that a transaction that maintains the integrity of the super schema would perform correctly when operating on an extended schema. The general condition guarantees that an object of a subclass when substituted where a superclass object is expected behaves just like an object of that superclass [17, 3]. Neither Java nor FJ nor FX can guarantee this property.

A database schema (Σ_B, E_B) is a consistent extension of a database schema (Σ_A, E_A) if the following conditions are satisfied:

- $\Sigma_B <: \Sigma_A$
- *The abstraction function absF* $: Mod(\Sigma_B) \to Mod(\Sigma_A)$ *has the property* $\mathcal{M}_B \models E_B \Rightarrow absF(\mathcal{M}_B) \models E_A$ *for all* Σ_B *models* $\mathcal{M}_B$.

In words, a legitimate instance of a consistent extension, when regarded as an instance of the super schema via the abstraction function, will satisfy all the constraints of the super schema, and will thus be a legitimate instance of the super schema. In this way the formalism provides a precise condition for semantically safe schema extension.

An example of a schema `ProjectManagement` that represents a consistent extension of the `Personnel` schema is given below.

```
public class ProjectManagement extends Personnel {
 public document contract [
    <ContractNo>[String]
    <Funds>[Float]
    <Customer>[String]
    <Leader>[String]
    forAll contract C, <ContractNo>[String] N
       ((N <> "" <- C<ContractNo>N))  ]
 public document degree [
    <Year>[String],
    <Degree>[<School>[String],
             <AbbDegree>[String]]]

 public interface Engineer extends Employee{
    degree* qualifications();
    Collection<Project> projects();
    forAll Engineer E (E.salary() > 100,000)}
 public interface Project{
    String projectId();
    Engineer leader();
    contract contract();
    Collection<Engineer> employees();
    forAll Project P, Engineer E
    (P.employees().contains(E) <- P.leader().equals(E))}

public class ProjectCollection
                   implements Collection<Project>{ ... }
protected projectCollection  dbProjects;
protected contract* dbContracts;
protected boolean establishProject(Project p);
protected boolean dissolveProject(Project p);

forAll  Engineer E, Project P, contract C
((dbEmployees.contains(E) <- dbProjects.contains(P),
                             P.engineers().contains(E)),
 (dbProjects.contains(P) <- dbEmployees.contains(E),
                            E.projects().contains(P)),
 (dbContracts.contains(C) <- dbProjects.contains(P),
                             P.contract() = C))
}
```

The results established in this section are based on an approach to constructing models for classes that represents an alternative to the models constructed in section 5. Models constructed in this section are also likely to be better suited for data models. Whenever a model for a class B is defined, the model for its superclass A (which might be *Object*) must be redefined (extended) to satisfy the condition $M_B \subseteq M_A$. This condition also requires that interpretation of methods of A must be extended accordingly so that the constraints E_A are satisfied, for the extension of A is to be consistent. We can define an abstraction function

$absF : Mod(\Sigma_B) \rightarrow Mod(\Sigma_A)$ so that

$absF(\mathcal{M}_B)$ is a family $\{M_C \mid C \in Sorts(B)\ and\ C \in Sorts(A)\}$.

Method definitions in the model $absF(\mathcal{M}_B)$ are defined in such a way that they agree with their definitions in the original model. The integrated type system, the constraint language and the associated model theory have been developed in a way to guarantee the following fundamental property:

Proposition 3
If (Σ_A, E_A) *and* (Σ_B, E_B) *are database schemas such that*

- $\Sigma_B <: \Sigma_A$ *and*
- $E_A \subseteq E_B$

Then (Σ_B, E_B) *is a consistent extension of* (Σ_A, E_A)*, i.e.,*
$\mathcal{M}_B \models E_B \ \Rightarrow \mathit{absF}(\mathcal{M}_B) \models E_A$.

Proof. Let $\mathcal{M}_B$ be a model of Σ_B. If $\mathcal{M}_B \models E_B$ then $\mathcal{M}_B \models E_A$ because $E_A \subseteq E_B$. We need $\mathit{absF}(\mathcal{M}_B) \models E_A$. Let e be a sentence in E_A. As we are only considering sentences that are universally quantified, strip the quantifiers off e to obtain an expression e', and let the free variables e' with their types be $C_1\ X_1, \ldots, C_m\ X_m$. Let θ be an assignment of values from $\mathit{absF}(\mathcal{M}_B)$ to the variables X_i that respects their types. Since $\mathit{absF}(\mathcal{M}_B)$ interprets the sorts in $\mathit{Sorts}(A)$ just as $\mathcal{M}_B$ does, such a θ is a valid variable assignment for $\mathcal{M}_B$. As the methods of Σ_A are interpreted in $\mathit{absF}(\mathcal{M}_B)$ to agree with their definitions in $\mathcal{M}_B$, $\mathit{eval}_\theta(e')$ in $\mathit{absF}(\mathcal{M}_B)$ will agree with $\mathit{eval}_\theta(e')$ in $\mathcal{M}_B$. But $\mathcal{M}_B \models e$, so $\mathit{eval}_\theta(e')$ in $\mathcal{M}_B$ is true. As θ was an arbitrary type respecting substitution of variables, we have $\mathit{absF}(\mathcal{M}_B) \models e$. Since e was any member of E_A, we have $\mathit{absF}(\mathcal{M}_B) \models E_A$, as desired.

9 Regular Document and Object Theories

A distinctive feature of this approach in comparison with FX is that regular expression types may be wrapped into object types. Since this paradigm is equipped with logic-based constraints, it allows specification of semantic conditions for wrapping and subtyping that cannot be enforced in Java.

Let Σ_T be a regular document type signature (which will in general include operations and predicates) and Σ_D the signature of a corresponding wrapper class D. Σ_D is equipped with two distinctive methods whose signatures are:
$\mathit{objectOf} : T \rightarrow D$ and $\mathit{valueOf} : D \rightarrow T$
where $\mathit{objectOf}$ denotes the constructor whose name in Java is D. This model theory is based on the following condition that governs the interpretation of the above two method signatures:
$f_{\mathit{valueOf}} f_{\mathit{objectOf}} = id_{M_T}$

If (Σ_T, E_T) is a theory of a regular document type T and (Σ_D, E_D) is the theory of its corresponding wrapper class then the wrapping function:
$\mathit{wrap}_{TD} : (\Sigma_T, E_T) \rightarrow (\Sigma_D, E_D)$
embeds the signature Σ_T into the signature Σ_D ($\Sigma_T \subseteq \Sigma_D$) and we also have $E_T \subseteq E_D$. The map is clearly a theory morphism which means that the following condition holds for all models $\mathcal{M}_D$ of Σ_D.

$\mathcal{M}_D \models E_D$ implies $\mathcal{M}_D \models \mathit{wrap}_{TD}(e)$ for all $e \in E_T$.

An example of a sentence in E_D is
$\mathit{forAll}\ T\ X, Y\ (\mathit{objectOf}(X).\mathit{equals}(\mathit{objectOf}(Y)) \leftarrow X = Y)$

The embedding morphism *wrap* induces a dual map
$\mathit{unWrap} : \mathit{Mod}(\Sigma_D) \rightarrow \mathit{Mod}(\Sigma_T)$

that maps a model of Σ_D to a model of Σ_T. This map forgets everything in the structure of $\mathcal{M}_D$ that models Σ_D but does not belong to Σ_T, and maps every object o in $\mathcal{M}_D$ to $valueOf(o)$.

The theory morphism condition guarantees that a wrapper object satisfies all the constraints of its corresponding regular document type. In addition, we also naturally require that $valueOf(o)$ satisfies all the constraints in E_T where o is an instance of D. This reasoning leads to the following general condition:

Proposition 4
The condition for semantically correct wrapping of a document type viewed as a theory (Σ_T, E_T) *into a theory of a document object type* (Σ_D, E_D) *is:*
$\mathcal{M}_D \models wrap(e)$ *iff* $unWrap(\mathcal{M}_D) \models e$ *for all* $e \in E_T$*.*
for any model $\mathcal{M}_D$ *of the theory* (Σ_D, E_D)*.*

Type signatures for document types and their corresponding object types participate in the subtyping relationships but in this approach they must also satisfy the semantic subtyping conditions. Formally, $\Sigma_B <: \Sigma_A$ is extended to a theory morphism $extend_{AB} : (\Sigma_A, E_A) \to (\Sigma_B, E_B)$.

So the theory of a wrapper class of a document type must incorporate two sets of constraints: those coming from its superclass and those coming from its regular document type. Let $T_2 <: T_1$ hold for document types so that $D_2 <: D_1$ holds for their corresponding wrapper classes. A sufficient semantic condition for wrapper class D_2 to both wrap T_2 and extend D_1 is expressed by the following commutative diagram of theory morphisms.

$$\begin{tikzcd}
(\Sigma_{T_1}, E_{T_1}) \arrow[r, "wrap_{T_1 D_1}"] & (\Sigma_{D_1}, E_{D_1}) \\
extend_{T_1 T_2} & extend_{D_1 D_2} \\
(\Sigma_{T_2}, E_{T_2}) \arrow[r, "wrap_{T_2 D_2}"] & (\Sigma_{D_2}, E_{D_2})
\end{tikzcd}$$

The commutativity of the diagram requires that the sorts, operations and sentences in (Σ_{T_1}, E_{T_1}) are mapped to the same corresponding entities in Σ_{D_2} regardless of the path taken, i.e., by first wrapping (Σ_{T_1}, E_{T_1}) into (Σ_{D_1}, E_{D_1}) and then extending (Σ_{D_1}, E_{D_1}) to (Σ_{D_2}, E_{D_2}) or constructing a subtype theory (Σ_{T_2}, E_{T_2}) of (Σ_{T_1}, E_{T_1}) and then wrapping it into (Σ_{D_2}, E_{D_2}).

Note that the equality of the composite morphisms $extend_{D_1 D_2} wrap_{T_1 D_1}$ and $wrap_{T_2 D_2} extend_{T_1 T_2}$ amounts to the following property:
$\mathcal{M}_{D_2} \models extend_{D_1 D_2}(wrap_{T_1 D_1}(e))$ and
$\mathcal{M}_{D_2} \models wrap_{T_2 D_2}(extend_{T_1 T_2}(e))$ for all $e \in E_{T_1}$.

If so, then the induced unwrapping and substitution maps should produce consistent (and in fact the same) results in accordance with the following diagram.

$$\begin{tikzcd}
\mathcal{M}_{D_2} \arrow[r, "unWrap_{D_2 T_2}"] & \mathcal{M}_{T_2} \\
abstractF & abstractF \\
\mathcal{M}_{D_1} \arrow[r, "unWrap_{D_1 T_1}"] & \mathcal{M}_{T_1}
\end{tikzcd}$$

The commutativity of the second diagram above should follow from the properties of the abstraction and unwrapping functions established in propositions

1, 2 and 3. However, the above semantic analysis has a major obstacle in regular document types. It is by no means obvious how to define a signature map underlying the theory morphism $extend_{T_1 T_2}$. The map $T_1 \rightarrow T_2$ is not an extension at all contrary to the situation in XML Schema [20] in which T_2 is an extension of T_1. This analysis indicates that a type system based on the type system of XML Schema might be a better candidate for the integration with a typed object-oriented paradigm in comparison with regular expression types.

10 Related Research

The object-oriented model presented in this paper extends a functional Java language FJ [13] with logic-based constraints. XML-like types are based on the work on regular expression types [12] for XDuce. The presented integrated paradigm builds on FX [11] which integrates FJ with regular expression types. Our approach to this integration is different and better suited for data models because it supports both complex XML-like values as well as XML-like object types within an object-oriented type system. This may be compared with the ODMG model [9] which features complex structured values in addition to objects. Another related piece of research on the integration of XML, an object-oriented language and features of data models within a unified type system is [18].

A distinctive and truly important feature is our extension of a desired unified type system with logic-based constraints that are not considered at all in FX or in [18]. This is also a major distinction in comparison with the industrial tools such as JDOM [15] and JAXB [14] that perform the transformations $XML \rightarrow Java \rightarrow XML$. These transformations are not necessarily governed by a typed schema, and certainly not with general semantic constraints for data.

Several research results are available at this point regarding XML constraints. XML Schema [20] has key and referential integrity constraints within a type system that features type extensions. Satisfaction and complexity results are available [6] for key and referential integrity constraints for DTDs [7]. Reasoning for more general key and functional dependencies for XML has been considered in [8]. Several classes of constraints (key, foreign key and inverse) are considered in [7] along with the complexity results for either specifying native XML documents or for preserving the semantics of data from other structured databases (relational or object-oriented). Neither of these papers considers constraints within the framework of a type system with features such as subtyping. Similar in spirit to our work is also the work on subsumption for XML types [16], but this work is unrelated to the object-oriented paradigm and it really does not address the issues of general constraints.

Several papers have considered the question of propagation of constraints between the relational and XML data models. [5] considers problems of inferring relational model functional dependencies for a transformation of an XML document instance. [4] describes an algorithm for constructing an XML document instance conforming to an XML schema with key and inclusion constraints from a relational database schema extension. Neither of these efforts addresses the issue of conversion of more general constraints, nor are they related to a typed or object-oriented paradigm. Similar remarks apply to other important results related to typechecking for XML, such as for example [19].

The model-theoretic nature of our approach, the type system, the logic-based constraints and the integration of XML-like types with a functional object-oriented paradigm make our work distinctive in comparison with the above cited papers. The model theory itself builds on the view of types as theories [10] that was developed fully for a typed object-oriented paradigm in [3]. This model theory has proved to be a suitable general framework for schema and data integration [2] as well as for the integration of objects, XML and database models [1]. A distinctive feature of this model theory is that it applies to a variety of logic paradigms and hence to a variety of constraint languages for either XML-related paradigms or the object-oriented paradigm or both.

11 Conclusions

The main contributions of this paper are:

- An approach to the integration of typed object-oriented and XML paradigms that is better suited to data models than the recently published alternatives.
- An extension of the integrated model with general logic-based constraints that are missing in alternative research results and industrial tools for this type of an integrated environment.
- A model theory of this sophisticated type and constraint based paradigm demonstrates that this integration is semantically well-founded.
- The generality of the developed model theory and its relative independence of a particular logic basis makes it applicable to a variety of approaches to XML (as well as object-oriented) constraints.
- The developed model theory offers general conditions for evolving the integrated schemas in such a way that both type and semantic conditions (the latter expressed by the integrity constraints) are satisfied.

Regular expression types were chosen for the proposed integration because of their limited complexity and their formal semantics. An additional important reason was that the integration of regular expression types with a Java-based functional language was already investigated from the viewpoint of an integrated type system [11]. This work is followed by our contribution in this paper that offers an alternative approach to this integration and extends it with constraints.

In retrospect, a suitably simplified XML Schema with its type system and already available constraint language appears to be a better candidate for the integration with the object-oriented paradigm. The type system of XML Schema features type extensions that are closer to object-oriented extensions via inheritance, and the constraint language, although very limited, is already available in XML Schema. Contrary to this, subtyping in regular expression types is structural [11], and is not based on explicit extensions as in the object-oriented paradigm. The type inference rules for this form of subtyping are not easy to define and are expensive to check [12] in spite of the allure of their semantic definition.

References

1. S. Alagić: Institutions: Integrating objects, XML and databases, *Information and Software Technology*, *44*, pp. 207 - 216, 2002.
2. S. Alagić and P. A. Bernstein, A model theory for generic schema management, Proceedings of DBPL '01 (Database Programming Languages), *Lecture Notes in Computer Science*, *2397*, pp. 228 - 246, 2002.
3. S. Alagić, S. Kouznetsova, Behavioral compatibility of self-typed theories. Proceedings of ECOOP 2002, *Lecture Notes in Computer Science 2374*, pp. 585-608, Springer, 2002.
4. M. Benedikt, C. Chee-Yong, W. Fan, J. Freire, and R. Rastog, Capturing both types and constraints in data integration, ACM SIGMOD Conference on Management of Data, 2003.
5. S. Davidson, W. Fan, C. Hara, and Qin Jing, Propagating XML constraints to relations, The 19th International Conference on Data Engineering (ICDE), 2003.
6. W. Fan and L. Libkin, On XML constraints in the presence of DTDs, Proceedings of ACM PODS, pp.114-125, 2001, also in *Journal of the ACM*, 49(3), pp. 368-406, 2002.
7. W. Fan and J. Simeon, Integrity constraints for XML, *Journal of Computer and System Sciences* 66, pp. 254-291, 2003.
8. P. Buneman, S. Davidson, W. Fan, C. Hara and W-C. Tan, Reasoning about keys for XML, Proceedings of DBPL '01 (Database Programming Languages), *Lecture Notes in Computer Science*, *2397*, pp.133 -148, 2002.
9. R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, 2000.
10. J. Goguen, Types as theories, in: G. M. Reed, A. W. Roscoe and R. F. Wachter, *Topology and Category Theory in Computer Science*, pp. 357-390, Clarendon Press, Oxford, 1991.
11. V. Gapeyev and B. Pierce, Regular object types, Proceedings of ECOOP 2003, *Lecture Notes in Computer Science 2743*, pp. 151-175, Springer, 2003.
12. H. Hosoya, J. Vouillon, and B. Pierce, Regular expression types for XML, Proceedings of ICFP, pp. 11-22, 2000.
13. A. Igarashi, B. Pierce, and P. Wadler, Featherweight Java: A minimal calculus for Java and GJ, Proceedings of OOPSLA 2001, and in *ACM Transactions on Programming Languages and Systems* 23(3), 2001.
14. JAXB documentation, `http://www.oasis-open.org/cover/jaxb.html`.
15. JDOM documentation, `http://www.jdom.org/`.
16. G. M. Kuper and J. Simeon, Subsumption for XML types, Proceedings of ICDT, *Lecture Notes in Computer Science 1973*, pp. 331-345, Springer, 2001.
17. B. Liskov and J. M. Wing, A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems*, *16*, pp. 1811-1841, 1994.
18. E. Meijer and W. Schulte, Unifying tables, objects and documents, Microsoft Research, 2003.
19. T. Milo, D. Suciu, and V. Vianu, Typechecking for XML transformers, *Journal of Computer and Systems Sciences*, 66(2003), pp. 66-07.
20. W3C: XML Schema, `http://www.w3c.org/XML/schema`.

M^2ORM^2: A Model for the Transparent Management of Relationally Persistent Objects

Luca Cabibbo and Roberto Porcelli

Dipartimento di Informatica e Automazione
Università degli studi Roma Tre
Via della Vasca Navale 79, 00146 Roma, Italy

Abstract. Object-oriented application development often involves storing application objects in a relational database. Sometimes it is desirable to develop the persistent classes and the relational database in an independent way, and to use an object persistent manager to connect them in a suitable way. This paper introduces M^2ORM^2, a model for describing meet-in-the-middle mappings between object schemas and relational schemas, to support the transparent management of object persistence by means of relational databases.

1 Introduction

Developing information systems involves various technologies, to be used in a combined and, possibly, synergic way. Relational database management systems provide an effective and efficient management of persistent, shared and transactional data [2]. Object-oriented tools and methods (programming languages and analysis and design methodologies) support the effective development of the application logic of complex information systems [16]. In practice, it is common to develop object applications with a layered architecture, containing at least an application logic layer and a persistence layer. *Persistent classes* are classes whose objects hold persistent data; they belong to the application logic layer, and are made persistent by means of code that connect them, in a suitable way, to the persistence layer. Object persistence can be achieved in several ways. In this paper, we will consider persistent objects managed by relational databases; that is, each application object is represented by means of tuples of a relational database.

Recently, various frameworks for the *transparent* management of object persistence have been implemented [18, 22]. By using them, the programmer manages persistent objects by means of standard API's such the ODMG ones [7], that is, the same way he would use objects in an object database. However, such frameworks can persist objects also by means of a relational database or files. Persistence is transparent to the programmer, since he does not know actual implementation details. The correspondence between objects and the underlying persistence support is managed by a software module, which can be implemented as a pre-compiler, a post-compiler, or an interpreter. Transparent persistence of

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 166–178, 2004.

objects can be achieved in various ways. In the *R/O mapping* approach (*Relation to Object mapping*, also called *reverse engineering*), persistent classes are automatically generated from a relational database. This way, the programmer populates the database by means of creations and modifications of objects from persistent classes. Then, persistent classes propagate such creations and modification to the underlying database. In the *O/R mapping* approach (*Object to Relation mapping*, also called *forward engineering*), starting from the classes that should be made persistent, the database is automatically generated, together with the code needed to propagate object persistence to the database.

A persistence management that is completely transparent is not always useful or possible. Often, an application should be developed that accesses an already existing relational database that is shared by many applications; this restricts the use of O/R mapping. Furthermore, the application logic designer needs to use all the features of the object model, without being constrained by the relational database schema; this restricts the use of R/O mapping. Luckily, there is a further way to persist objects, which allows to manage the cases in which the application logic and the database have been developed and evolve in an independent way. The *meet-in-the-middle* approach assumes that the persistent classes and the database are designed and implemented separately. In this case, the correspondences between persistent classes and the relational database should be given, possibly described in a declarative way. These correspondences describes a "meet in the middle" between the two schemas and are used by the persistence manager to let the objects persist by means of the database. The meet-in-the-middle approach is very versatile, since modifications in persistent classes and/or in the relational database can be managed by simply redefining the correspondences.

Unfortunately, existing systems support the meet-in-the-middle approach only in a limited way. Several object persistence managers are indeed based on the O/R and R/O approaches and, essentially, they allow to manage only correspondences between similar structures (e.g., the objects of a single class with the tuples of a single relation); such systems may permit a meet between the two schemas, but only as a local tuning activity, after the schema translation from one model to the other. Limitations of existing systems are motivated by the difficulties in reasoning about complex correspondences to avoid anomalies and inconsistencies.

This paper introduces M^{2}ORM^2 (an acronym for *Meet-in-the-Middle Object/Relational Mapping Model*), a model to describe mappings (that is, correspondences) between object schemas and relational schemas, to support the transparent management of object persistence based on the meet-in-the-middle approach. With respect to currently available systems, M^{2}ORM^2 allows for more possibilities to meet schemas. Rather than considering only correspondences between single classes and single relations, M^{2}ORM^2 allows to express complex correspondences between clusters of related classes (intuitively, each representing a single concept) and clusters of related relations. Furthermore, it is possible to express correspondences describing relationships between clusters. With re-

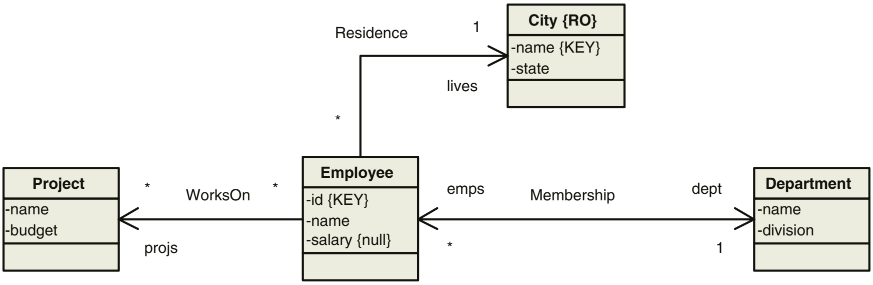


Fig. 1. An object schema.

spect to other proposals, M^{2}ORM^2 considers specific details of the object and relational data models, to identify as many ways to meet schemas as possible.

The main contribution of this paper is the introduction of M^{2}ORM^2. Furthermore, a discussion is started on the correctness of mappings, by identifying some correctness conditions.

Section 2 proposes terminology and notation used to describe object schemas and relational schemas. Section 3 introduces M^{2}ORM^2, to describe mappings between schemas, together with an example. Section 4 discusses the problem of mapping correctness. Finally, Section 5 discusses the effectiveness of the model, by comparing it with related proposals.

2 Object Schemas and Relational Schemas

This section presents briefly the data models (an object model and the relational model) and the terminology used in this paper.

The *object model* we consider is a non-nested semantic data model (with structural features, but without behavioral ones). We have in mind a Java-like object model, formalized as a simplified version of the ODMG model [7] and of UML [5].

At the schema level, a *class* describes a set of *objects* having the same structural properties. Each class has a set of *attributes* associated with it, each having a name and a type; in this paper we make the simplifying hypothesis that all class attributes are of a same simple type, e.g., strings. An *association* describes a binary relation between a pair of classes (in reality, between the objects of such classes). A pair of classes can be in a *generalization/specialization relationship*; this implies attribute and association inheritance from the superclass to the subclass. We consider single inheritance only. An *object schema* is a set of classes, related by generalization/specialization hierarchies, and associations among such classes. Figure 1 shows a sample object schema.

At the instance level, a class is a set of *objects*. More specifically, each object is instance of exactly one class. However, because of generalization/specialization relationships, an object may belong to more than one class: the one from which it has been instantiated and all of its superclasses. Each object is associated with an *oid*, an unique identifier that allows to reference the object. The *state* of an object is given by the set of values that its attributes hold in a certain moment. An association is a set of *links*; each link describes a relationship between a pair of objects.

This paper takes into consideration the following integrity constraints. Class attributes can have null value; an attribute whose value cannot be null is said to be *not null*. Sometimes it is useful to search an object in a class on the basis of the value of some attributes; in this case, the attributes identifying the objects are called *key attributes* and the class is said to be *with key*; otherwise, the class is *without key*. Key attributes should be not null. A *read-only class* is a class from which it is not allowed to instantiate new persistent objects and to modify and to delete already existing objects. In an application, read-only classes are useful to access information generated by other applications that cannot be modified by this application. In Figure 1, key attributes are denoted by the constraint {*KEY*}, and attributes that can be null by means of the constraint {*null*}; constraint {*RO*} denotes read-only classes.

For associations we consider multiplicity and navigability constraints. A *role* is an end of an association, that is, a class involved in the association. Roles have name, navigability, and multiplicity. The *multiplicity* of a role in an association denotes how many objects (at least and at most) of that class can be linked to each object of the class on the other role of the association. The *navigability* of a role denotes the possibility to reach the objects of that class by means of links from objects of the other role of the association.

In the *relational model* [2], at the schema level a *relation* describes a set of tuples. A relation schema is a set of *attributes*, each with a name and a type; in this paper, we assume that all relation attributes are of a simple type, e.g., strings. A *relational schema* is a set of relations. At the instance level, a relation is a set of *tuples* over the attributes of the relation.

This paper takes into consideration the following integrity constraints. Attributes can be or not be *not null*. Each relation has a *key*, that is, a non-empty set of attributes that allows the identification of the tuples of the relation. A *key attribute* is an attribute that belongs to a key; key attributes should be not null. Sometimes relations are identified by means of *artificial keys* (or *surrogates*), rather than by means of *natural keys* (that is, keys based on attributes having a natural semantics). In a relation with artificial key, the insertion of a new tuple involves the generation of a new artificial key; the DBMS is usually responsible of this generation. For *referential constraint* (or *foreign key*) we mean a non-empty set of attributes of a relation used to reference tuples of another relation.

Figure 2 shows a sample relational schema. Attribute forming natural keys are denoted by the constraint {*NK*}, and those forming artificial keys by the constraint {*AK*}. Referential constraints are denoted by arrows and implemented by means of attributes with the constraint {*FK*}. The constraint {*null*} denotes attributes that can be null.

We assume that programmers manage object schemas only in a programmatic way. In practice, objects and links are manipulated by means of *CRUD*

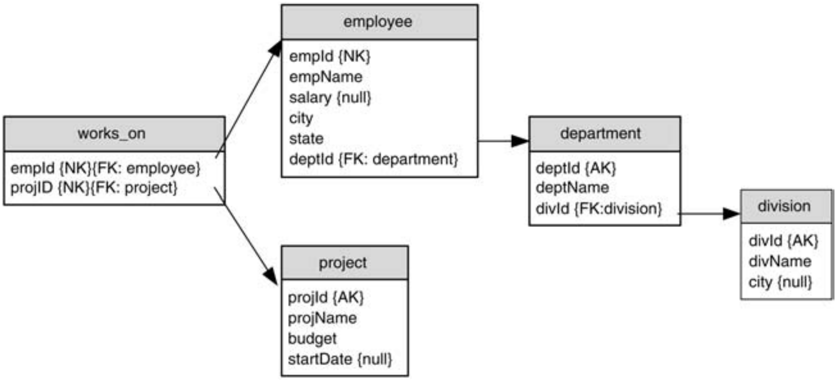


Fig. 2. A relational schema.

```
/* increase the salary of Employee #123 */
PersistenceManager pm = new PersistenceManager(...);
Transaction tx = pm.newTransaction();
tx.begin();
Employee e = (Employee) pm.getObjectById( new EmployeeId("123") );
e.setSalary( e.getSalary()*1.1 );
tx.commit();
```

Fig. 3. Persistent objects are managed in a programmatic way.

operations (*Create*, *Read*, *Update*, *Delete*), that allow the creation of persistent objects, the reading of persistence objects (that is, the unique search of an object based on its key), as well as the persistent modification and deletion of objects. Moreover, navigation, formation, breaking, and modification of persistent links between objects is allowed. Reading is meaningful only for classes with key. Creation, modification, and deletion is not meaningful for read-only classes. For example, Figure 3 shows a possible fragment of code to increase the salary of an employee, with respect to the scheme of Figure 1.

In correspondence to such programmatic manipulations of an object schema, a meet-in-the-middle-based object persistence manager should translate CRUD operations on objects and links into operation over an underlying relational database. This translation should happen in an automatic way, on the basis of a suitable mapping between the object schema and the relational schema, as described in the next section.

3 A Model for Object/Relational Schema Mappings

M^{2}ORM^2 (an acronym for *Meet-in-the-Middle Object/Relational Mapping Model*) is a model for describing mappings among object schemas and relational schemas, to support the transparent management of relationally persistent objects based on the meet-in-the-middle approach. In this paper, we assume that an object schema and a relational schema have been independently developed. In particular, the relational schema could be (partially) denormalized for efficiency reasons; the object schema could be denormalized as well, that is, it may contain "coarse grain" objects.

In this paper, a M^2ORM^2 mapping is described by means of a multi-graph (that is, by a set of nodes and a set of arcs, with the observation that several arcs can connect a same pair of nodes). Each node describes a correspondence between a set of classes and a set of relations. Each arc describes a relationship between the correspondences described by means of a pair of nodes. Intuitively, using the Entity-Relationship terminology [3], each node represents an entity (which can be denormalized in one of the schemas) and each arc represents a binary relationship or a generalization/specialization relationship between two entities. In practice, we envision the possibility to describe M^2ORM^2 mappings as structured (e.g., XML) text files, as it happens in current systems.

A *class cluster* (or *c-cluster*) comprises a non-empty set of classes and a set of associations and/or generalization/specialization relationships among such classes[1]. In a c-cluster, one of the classes should be selected as *primary class* of the c-cluster; the other classes are *secondary* ones. The associations of the node should be, directly or indirectly, all of type one-to-one or one-to-many from the primary class to secondary classes. Intuitively, such associations should relate each object of the primary class with at most an object from each of the secondary classes.

A *relation cluster* (or *r-cluster*) comprises a non-empty set of relations, together with referential constraints among them. In an r-cluster, one of the relations should be selected as the *primary relation* of the r-cluster; the other relations, called *secondary*, should be referenced, directly or indirectly, from the primary relation of the r-cluster. Intuitively, each tuple of the primary relation should be associated with at most a tuple in each of the secondary relations through the referential constraints of the r-cluster.

Each *node* of a mapping describes the correspondence between a c-cluster and an r-cluster, by means of information about correspondences among their elements (classes, relations, attributes, associations, and generalization/specialization relationships). Specifically, in a node the correspondence among c-cluster elements and r-cluster ones are described by means of *attribute correspondences*, each of which involves an attribute of a class and an attribute of a relation. In M^2ORM^2, there are three kinds of nodes, to let a class correspond with a relation, a class with multiple relations, and multiple classes with a relation. Currently, the model does not permit that multiple classes correspond with multiple relations; however, most complex correspondences of this kind can be represented by means of arcs.

[1] More precisely, each element of a c-cluster is associated with a class in the object schema. Therefore, it is possible that a class takes part to more c-clusters, or that it takes part more the once in a same c-cluster. A similar consideration applies to r-clusters, that will be introduced shortly.

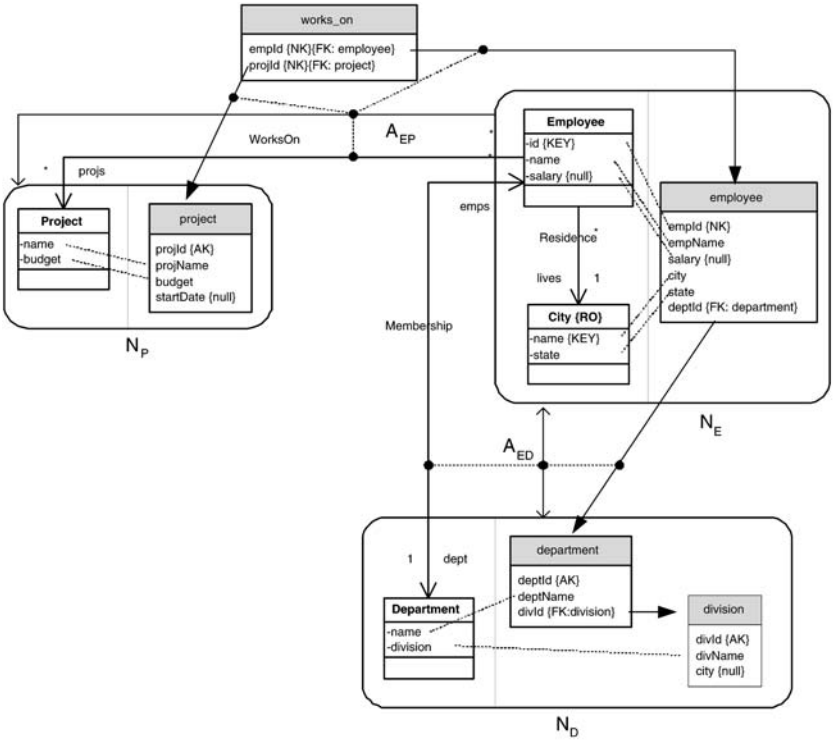


Fig. 4. A M^2ORM^2 mapping between the schemas of Figures 1 and 2.

Figure 4 shows a M^2ORM^2 mapping between the schemas of Figures 1 and 2, based on three nodes: a node N_P, describing the correspondence between class *Project* and relation *project*; a node N_E, describing the correspondence between classes *Employee* and *City* and relation *employee*; and a node N_D, describing the correspondence between class *Department* and relations *department* and *division.*

Node N_P describes the correspondence between a class (*Project*) and a relation (*project*); it is clear that both the class and the relation are primary in the node. The correspondence between them is based on the attribute correspondences *(Project.name, project.projName)* and *(Project.budget, project.budget)*; these are depicted as dotted lines in Figure 4. Node N_P describes a *total* correspondence between a class and a relation: each object of the class is represented by means of a tuple of the relation. This correspondence involves a class without key and a relation with artificial key. All the attributes of the class are involved in the correspondence; however, not all the attributes of the relation are involved in it. This correspondence allows to manage, for example, the creation of a new persistent *Project* object by means of the insertion of a new tuple in *project*; in this tuple, the values for *project.projName* and *project.budget* come from the object attributes *Project.name* and *Project.budget*, on the basis of attribute correspondences; moreover, the value for *project.projId* is automatically generated and the value for *project.startDate* remains null.

Node N_E describes the correspondence between two classes (*Employee* and *City*) linked by an association (*Residence*) and a relation (*employee*). In this node, *Employee* is the primary class. Intuitively, each object of the primary class *Employee* can be associated (by means of the association *Residence*) with an object of the secondary class *City*. Each tuple of *employee* represents an *Employee* object together with the *City* object related to it, and also the link of type *Residence* between these two objects. In this node, in which the primary class *Employee* is with key, the correspondence is based on the correspondence between the key attributes of the primary class and of the primary relation *(Employee.id, employee.empId)* together with the attribute correspondences *(Employee.name, employee.empName)*, *(Employee.salary, employee.salary)*, *(City.name, employee.city)*, and *(City.state, employee.state)*. For example, the creation of a new *Employee* object, linked to an existing *City* object, is managed by the insertion of a new tuple in *employee*. (The management of the membership of the employee in a department is described later.)

Finally, node N_D describes the correspondence between a class (*Department*) and two relations (*department* and *division*); the primary relation is *department*. Intuitively, each *Department* object is represented by a tuple of *department* and a tuple of *division*, related by means of a referential constraint. The correspondence is based on the attribute correspondences *(Department.name, department.deptName)* and *(Department.division, division.divName)*. The latter correspondence is meaningful with respect to the referential constraint between the primary relation *department* and the secondary relation *division* of the r-cluster. Indeed, this node allows to represent in the mapping the two relations together with the referential constraint between them. For example, the creation of a new *Department* object is managed by the insertion of a new tuple in *division* (with the generation of an artificial key) and of a new tuple in *department* (with the generation of another artificial key).

In general, not every mapping is correct. For example, if the primary class of a node is with key, then it is necessary that its key attributes correspond to the key attributes of the primary relation of the node. However, if the primary class is without key, then the primary relation should be with artificial key. Correctness of mappings will be discussed briefly in Section 4.

A M^{2}ORM^2 mapping can contain arcs. Intuitively, arcs allow to represent correspondences and elements which cannot be represented by means of nodes; specifically, further associations, generalization/specialization relationships, referential constraints, and some further relations. Each arc describes a relationship between a pair of nodes, and can be a binary relationship (of type one-to-one, one-to-many, or many-to-many) or a generalization/specialization relationship.

An arc involves a class correspondence and a relation correspondence. A *class correspondence* describes the correspondence between the primary classes of the c-clusters of the nodes connected by the arc, and it is based either on the navigable roles of an association between the classes or on a generalization/specialization relationship. In practice, the roles are implemented by means of reference attributes (for the to-one navigability) and/or collections of reference attributes (for the to-many navigability). If the association is unidirectional, then a single attribute is involved; otherwise, if it is bidirectional, there are two attributes involved. A *relation correspondence* describes the correspondence between the primary relations of the r-clusters of the nodes connected by the arc, and it is based on the set of attributes that implement the relationship between the two relations by means of referential constraints; further relations can be involved. An arc groups a class correspondence and a relation correspondence, representing a one-to-one, one-to-many, or many-to-many binary relationship or a generalization/specialization relationship between the instances represented by a pair of nodes.

For example, the mapping shown in Figure 4 contains two arcs: an arc A_{ED} for the one-to-many association between *Employee* and *Department* and an arc A_{EP} for the many-to-many association between *Employee* and *Project*.

Arc A_{ED} between nodes N_E and N_D describes the one-to-many relationship between employees and departments. This relationship is represented by means of the correspondence *[Employee.dept, Department.emps]* between the classes *Employee* and *Department* and by means of the correspondence *[employee.deptId]* between the relation *employee* and *department*. Intuitively, this arc lets the association *Membership* between *Employee* and *Department* correspond with the referential constraint between *employee* and *department*. For example, if an *Employee* object e changes its membership (that is, its *dept* reference attribute) to a *Department* object d, then the *deptId* attribute of the *employee* tuple representing e is updated to reference the *department* tuple representing d.

Arc A_{EP} between nodes N_E and N_P describes the many-to-many relationship between employees and projects. This arc is based on the (unidirectional) correspondence *[Employee.projs]* between classes *Employee* and *Project* and on the correspondence *[works_on.empId, works_on.projId]* between relations *employee* and *projects*. In practice, this arc lets the association *WorksOn* between *Employee* and *Project* correspond with the referential constraints between *employee* and *project* stored in the tuple of the relation *works_on*. For example, if a *WorksOn* link between an *Employee* object e and a *Project* object p is broken, then the *works_on* tuple representing this link is deleted.

To discuss arcs representing generalization/specialization relationships, consider the schemas shown in Figure 5. A mapping between them can be defined by means of two nodes (for persons and students) and a generalization/specialization arc connecting them. In the arc, the generalization/specialization between the two classes corresponds with the referential constraint from the primary key of *student* to *person*.

In practice, there are various ways to represent a generalization/specialization hierarchy by means of a relational database [3]. Until now, $\mathrm{M}^2\mathrm{ORM}^2$ can only represent the one that essentially maps each class of the hierarchy with a different relation.

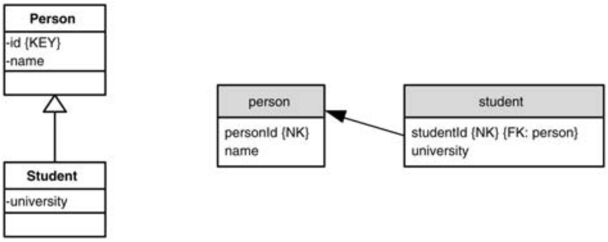


Fig. 5. A schema involving a generalization/specialization relationship.

4 Correctness of Mappings

In the previous section, M^{2}ORM^2 has been used as a syntactical tool to represent mappings. In reality, mappings have semantics as well, to describe how CRUD operations on objects and links of the object schema can be realized by means of operations on the relations of the relational schema. However, not every mapping that can be described using M^{2}ORM^2 is correct. Intuitively, a mapping is *correct* if it supports, in an effective way, the management of CRUD operations on objects and links by means of the relational schema. Otherwise, a mapping is *incorrect* if operations on objects and links can give rise to anomalies. In the following of this section we identify classes of anomalies caused by incorrect mappings, and some categories of conditions suitable for studying the correctness of M^{2}ORM^2 mappings.

We consider a sequence of CRUD operations on objects and links that, globally, transforms a valid instance of the object schema into another valid instance, that is, in which all the integrity constraints imposed by the schema are satisfied. An incorrect mapping can give rise to the following anomalies during the execution of such sequence of operations:

- *creation anomalies* happen when the relational schema is not able to manage the creation of a new object in some class (not a read-only class); for example, a creation anomaly happens if the class is not represented in the mapping or if some of its attributes are not in a suitable correspondence with attributes of the relational schema (including erroneous correspondences involving not null attributes and key attributes);
- *reading anomalies* are related to the inability to identify uniquely an object from a class with key; for example, a reading anomaly happens if there are erroneous correspondences involving key attributes of classes and relations;
- *update anomalies* happen if it is not possible to manage the modification of attributes of an object (belonging to non read-only class);
- *deletion anomalies* happen if it is not possible to manage the deletion of an object (belonging to non read-only class).

Similarly, there are anomalies related to the formation of links, and also to their navigation, modification, and breaking.

There are two main causes for anomalies: *incorrect correspondences between elements* and *incorrect representation of integrity constraints*.

An example of the former case happens if the correspondences among elements are incomplete, that is, if there are elements of the object schema (classes, associations, attributes) that are not in correspondence with elements of the relational schema. Indeed, each class must belong to at least one node, and the attributes of classes should occur in at least one attribute correspondence. However, relational elements must obey to different conditions; for example, it is possible that a relation or some of its attributes do not partecipate in the mapping.

Inconsistencies in the representation of integrity constraints can happen in several different ways. For example, with respect to key constraints, it is necessary that key attributes of primary classes with key are in correspondence with key attributes of primary relations with natural key. Furthermore, primary classes without key should correspond to relations with artificial key. As a further example, concerning referential constraints, in a node that lets a class correspond with more than a relation, non-key attributes of the class should correspond to non-key attributes of relations, and referential constraints towards secondary relations should be implemented by means of artificial keys.

The conditions we have just described are *necessary* conditions for the correctness of $\mathrm{M}^2\mathrm{ORM}^2$ mappings. However, in practice (i.e., for the implementation of a persistence manager based on $\mathrm{M}^2\mathrm{ORM}^2$) *sufficient* conditions should be fixed, to manage mappings in an effective way. Current systems suffer from several limitations, since conditions they are based on are very restrictive. One of the main goal of this research is to identify conditions that are as permissive as possible with respect to which mappings are semantically meaningful.

5 Discussion

The "professional" literature on databases and on pattern languages is rich of works on how to manage persistent classes by means of relational databases [1, 6, 10, 15]. Most of these works describes the O/R mapping, a kind of logical design [3] starting from an object schema rather than from an entity-relationship schema. But, as we already said in the Introduction, O/R mapping assists the definition of a database supporting a single application, and not *shared* among several applications, being this one of the main motivations for using a DBMS as the manager of persistent data.

This topic has been investigated by the scientific literature of the database community as well. Persistence [14] is a system supporting the O/R mapping. Gateway [19] is a system for managing persistent objects by means of a relational database based on the meet-in-the-middle approach; however, the paper does not give details on the mapping model used. EBO [20] is an R/O mapping system, featuring some functionalities of meet-in-the-middle; also in this case, the mapping model has not been described.

The problem of describing and analyzing schema mappings is also of interest in the areas of data transformations [8] and data integration [9]. The notion of

Table 1. Comparison with other models and tools.

Feature	JDO	OJB	JRELAY	JDX	M^2ORM^2
O/R mapping (forward engineering)	yes	yes	yes	yes	no
R/O mapping (reverse engineering)	n/a	yes	yes	yes	no
Meet in the middle	n/a	yes	yes	yes	yes
One class/one relation	n/a	yes	yes	yes	yes
One class/many relations with referential constraints	n/a	yes[a]	yes	no	yes
One class/many relations with vertical partitioning	n/a	no	no	no	yes[b]
Many classes/one relation	n/a	no	no	yes	yes
Many classes/many relations	n/a	no	no	no	no
Classes with key	yes	yes	yes	yes	yes
Classes without key	yes	no	yes	no	yes
Read-only classes	no	no	no	no	yes
Attribute of a class mapped into several relations	n/a	no	no	no	yes
One-to-one relationship	n/a	yes	yes	yes	yes
One-to-many relationship	n/a	yes	yes	yes	yes
Many-to-many relationship without attributes	n/a	yes	yes	no	yes
Many-to-many relationship with attributes	n/a	no	no	no	yes[b]
Many-to-many relationship with selective attributes	n/a	no	no	no	yes[b]
Generalization/specialization hierarchies	yes	yes	yes	yes	yes[c]

[a] Yes, but not implemented.
[b] Yes, but not discussed in this paper.
[c] Yes, but discussed only partially in this paper.

mapping used in this paper is inspired from the one proposed in the context of model management [4].

Table 1 compares M^2ORM^2 with some standards and commercial and/or open source systems for the transparent management of persistent objects. Java Data Objects (JDO) [11] is a standard for the transparent management of Java objects. OJB [18] is an Apache open source project, implementing the ODMG and the JDO API's; in practice, it allows to manage persistent objects as they were stored in an ODMG object database [7]. JDX [12] features O/R mapping functionalities similar to OJB ones; however, persistent objects are manipulated by means of proprietary API's. JRELAY [13] is a JDO implementaion, implementing the three mapping approaches, but with significative limitations. From the table, it is possible to see that M^2ORM^2 provides the same features offered by current systems, plus more.

The management of generalization/specialization hierarchies needs a special comment. Until now, M^2ORM^2 supports them only in a limited way, but still comparable to the support given by other systems. In future, we plan to manage generalization/specialization relationships in a more complete way.

As future work, we plan the implementation of a framework for the management of persistent Java objects based on M^{2}ORM^2. From a theoretical perspective, we plan to study several extensions to the model (specifically, the management of generalization/specialization hierarchies, transient classes and attributes, further integrity constraints, and schematic heterogeneities [17]) but also to give more precise characterizations of correct M^{2}ORM^2 mappings.

References

1. S.W. Ambler. The fundamentals of mapping objects to relational databases. White Paper, `http://www.agiledata.org`, 2003.
2. P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems. Concepts, Languages and Architectures.* McGraw-Hill, 1999.
3. C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database Design, an Entity-Relationship Approach.* Benjamin-Cummings, 1992.
4. P.A. Bernstein, A.Y. Halevy, and R.A. Pottinger. A vision for the management of complex models. *ACM Sigmod Record*, 29(4):55–63, 2000.
5. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.
6. K. Brown and B.G. Whitenack. Crossing Chasms: A pattern language for object-RDBMS integration. In *Pattern Languages of Program Design 2*, 1996.
7. R.G.G. Cattell et al. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, 2000.
8. Data Transformations. S.ı. of the *IEEE Bull. on Data Engineering*, 22(1), 1999.
9. Integration management. S.ı. of the *IEEE Bull. on Data Engineering*, 25(3), 2002.
10. M.L. Fussell. Foundations of object relational mapping. White Paper, `http://www.chimu.com`, 1997.
11. Java Data Objects. `http://www.jdocentral.com`.
12. JDX. `http://www.softwaretree.com/`.
13. JRELAY. `http://www.objectindustries.com/`.
14. A.M. Keller, R. Jensen, and S. Agrawal. Persistence Software: Bridging object-oriented programming and relational databases. In *ACM SIGMOD International Conf. on Management of Data*, pages 523–528, 1993.
15. W. Keller. Mapping object to tables: A pattern language. In *European Conf. on Pattern Languages of Programming*, 1997.
16. C. Larman. *Applying UML and Patterns. An introduction to object-oriented analysis and design and the Unified Process.* Prentice Hall PTR, 2002.
17. R.J. Miller. Using schematically heterogeneous structures. In *ACM SIGMOD International Conf. on Management of Data*, pages 189–200, 1998.
18. ObJect relational Bridge. `http://db.apache.org/ojb/`.
19. J.A. Orenstein and D.N. Kamber. Accessing a relational database through an object-oriented database interface. In *21st Int. Conf. on VLDB*, 702–705, 1995.
20. J.A. Orenstein. Supporting retrievals and updates in an object/relational mapping system. *IEEE Bull. on Data Engineering*, 20(1):50–54, 1999.
21. E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, 2001.
22. Torque. `http://db.apache.org/torque/`.

Using XQuery for Flat-File Based Scientific Datasets*

Xiaogang Li and Gagan Agrawal

Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210
{xgli,agrawal}@cis.ohio-state.edu

Abstract. XQuery is a recently developed query language for XML datasets. In this paper, we focus on the use of XQuery and other XML technologies for flat-file based scientific datasets. Traditionally, complex and domain-specific data layouts have complicated the processing of large datasets arising from scientific applications. The use of XML schemas and XQuery's high-level structure can simplify the analysis on these datasets.
Though scientific data processing applications can be conveniently represented in XQuery, compiling them to achieve efficient execution involves a number of challenges. These are, 1) analysis of recursive functions to identify reduction computations involving only associative and commutative operations, 2) replacement of recursive functions with iterative constructs, 3) application of data-centric transformations on the structure of XQuery, and 4) translation of XQuery processing to an imperative language like C/C++, which is required for using a middleware that offers low-level data access functionality. This paper describes our solutions towards these problems and demonstrates significant benefits from the transformations we have developed.

1 Introduction

XQuery is a recently developed language for querying and processing XML datasets. We have initiated an effort for supporting the use of XQuery for applications that process scientific datasets. The motivation for our work is three-fold. First, distribution and use of data comprising scientific images and experimental or simulation results has traditionally been hindered by complex and specialized formats. XML is already being used for standerizing the distribution of data in many domains, and this trend can only be expected to continue. Thus, it is natural to use XQuery for specifying processing on data. Second, as we show in this paper, XQuery can significantly simplify the specification of processing on complex scientific datasets. Third, we believe that scientific data processing applications provide a new class of interesting benchmarks for XQuery. Most of the existing applications targeted for XQuery involve searching for records or documents with certain characteristics [13], and the existing implementation efforts have typically targeted these. In contrast, scientific data processing applications involve non-trivial computation.

* This work was supported by NSF grant ACR-9982087, NSF CAREER award ACR-9733520, NSF grant ACR-0130437 and NSF grant ACI- 0203846. The equipment used for the experiments reported here was purchased under the grant EIA-9986052.

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 179–194, 2004.

We have particularly focused on applications that process multi-dimensional datasets, and involve *generalized reductions*. Analysis and processing of very large multi-dimensional scientific datasets (i.e. where data items are associated with points in a multidimensional attribute space) is an important component of science and engineering. Examples of these datasets include raw and processed sensor data from satellites, output from hydrodynamics and chemical transport simulations, and archives of medical images[1]. Our study of a variety of scientific data-intensive applications has shown that generalized reduction operations are very common in the processing structure. Processing for generalized reductions consist of three main steps. (1) Retrieving data items of interest, (2) Applying application-specific transformation operations on the retrieved input items, and, (3) mapping the input items to output items and aggregating, in some application specific way, all the input items that map to the same output data item. Most importantly, aggregation operations involve *commutative* and *associative* operations, i.e., the correctness of the output data values does not depend on the order input data items are aggregated.

While such scientific data processing applications can be conveniently coded in XQuery, compiling them to achieve efficient execution involves a number of challenges. Because XQuery is a functional language, the only practical way for specifying reduction computations is using recursion. This, and some of the other related features of XQuery, lead to the following compiler analysis and restructuring challenges:

- Analysis of recursive functions to identify reduction computations involving only associative and commutative operations.
- Replacement of recursive functions with iterative constructs.
- Application of data-centric transformations on the structure of XQuery.
- Translation of XQuery processing to an imperative language like C/C++, which is required for using a middleware that offers low-level functionality. Particularly, the challenge is to deduce the types of data-structures to be used in the imperative language.

In this paper, we report our solutions towards the above problems. By implementing the techniques in a compiler and generating code for a runtime system called Active Data Repository (ADR) [3], we are able to achieve efficient processing of disk-resident datasets.

2 XQuery Language

As stated previously, XQuery is a language currently being developed by the World Wide Web Consortium (W3C). It is designed to be a language in which queries are concise and easily understood, and to be flexible enough to query a broad spectrum of information sources, including both databases and documents.

XQuery is a functional language. The basic building block is an *expression*. Two classes of expressions important for our work are as follows:

- FLWR expressions, which support iteration and binding of variables to intermediate results. FLWR stands for the keywords*for*, *let*, *where*, and *return*.

– Unordered expressions, which use the keyword *unordered*. The unordered expression takes any sequence of items as its argument, and returns the same sequence of items in a nondeterministic order.

```
for $d in document(”depts.xml”)//deptno
  let $e := document(”emps.xml”)//emp[deptno = $d]
      where count($e) >= 10
  return
    <big-dept>
      {
        $d,
        <headcount> { count($e) } </headcount>,
        <avgsal> {avg($e/salary)} </avgsal>
      }
    </big-dept>
```

Fig. 1. An Example Illustrating XQuery's FLWR Expressions

We illustrate the XQuery language and the *for*, *let*, *where*, and *return* expressions by an example, shown in Figure 1. In this example, two XML documents, *depts.xml* and *emps.xml* are processed to create a new document, which lists all departments with ten or more employees, and also lists the average salary of employees in each such department.

In XQuery, a *for* clause contains one or more variables, each with an associated expression. The simplest form of *for* expression, such as the one used in the example here, contains only one variable and an associated expression. The evaluation of the expression typically results in a sequence. The *for* clause results in a loop being executed, in which the variable is bound to each item from the resulting sequence in turn. In our example, the sequence of distinct department numbers is created from the document *depts.xml*, and the loop iterates over each distinct department number.

A *let* clause also contains one or more variables, each with an associated expression. However, each variable is bound to the result of the associated expression, without iteration. In our example, the *let* expression results in the variable $e being bound to the set or sequence of employees that belong to the department $d. The subsequent operations on $e apply to such sequence. For example, $count(\$e)$ determines the length of this sequence.

A *where* clause serves as a filter for the tuples of variable bindings generated by the *for* and *let* clauses. The expression is evaluated once for each of these tuples. If the resulting value is true, the tuple is retained, otherwise, it is discarded. A *return* clause is used to create an XML record after processing one iteration of the *for* loop. The details of the syntax are not important for our presentation.

To illustrate the use of *unordered*, a modification of the example in Figure 1 is presented in Figure 2. By enclosing the *for* loop inside the *unordered* expression, we are not enforcing any order on the execution of the iterations in the *for* loop, and in generation of the results. Without the use of *unordered*, the departments need to be processed in the order in which they occur in the document *depts.xml*. However, when *unordered* is used, the system is allowed to choose the order in which they are processed, or even process the query in parallel.

```
unordered(
  for $d in document(”depts.xml”)//deptno
    let $e := document(”emps.xml”)//emp[deptno = $d]
        where count($e) >= 10
    return
      <big-dept>
        {
          $d,
          <headcount> { count($e) } </headcount>,
          <avgsal> {avg($e/salary)} </avgsal>
        }
      </big-dept>
)
```

Fig. 2. An Example Using XQuery's Unordered Expression

3 Scientific Data Processing Applications in XQuery

In this section, we describe a motivating scientific data processing application, *satellite data processing* [3], and show how it can be expressed using XQuery.

3.1 Satellite Data Processing

The first application we focus on involves processing the data collected from satellites and creating composite images. A satellite orbiting the Earth collects data as a sequence of *blocks*. The satellite contains sensors for five different bands. The measurements produced by the satellite are short values (16 bits) for each band.

The typical computation on this satellite data is as follows. A portion of Earth is specified through latitudes and longitudes of end points. A time range (typically 10 days to one year) is also specified. For any point on the Earth within the specified area, all available pixels within that time period are scanned and an application dependent output value is computed. To produce such a value, the application will perform computation on the input bands to produce one output value for each input value, and then the multiple output values for the same point on the planet are combined by a reduction operation. For instance, the Normalized Difference Vegetation Index (ndvi) is computed

```
unordered(
   for $i in ($minx to $maxx)
      for $j in ($miny to $maxy)
         let $p := document(”satellite.xml”)/data/pixel
               where(( $p/x = $i) and ($p/x = $j ))
         return
            <pixel>
               <latitute> {$i} </latitute>
               <longtitute> {$j} </longtitute>
               <summary> {accumulate($p)} </summary>
            </pixel>
)

define function accumulate ($p)
      as double
{
   let $inp := item-at($p,1 )
   let $NVDI := ( ($inp/band1 - $inp/band0) div
            ($inp/band1 + $inp/band0)+1) * 512
   return
      if( empty($p) )
      then 0
      else { max($NVDI, accumulate(subsequence($p,2))) }
}
```

Fig. 3. Satellite Data Processing Expressed in XQuery

based on bands one and two, and correlates to the “greenness” of the position at the surface of the Earth. Combining multiple ndvi values consists of execution a max operation over all of them, or finding the “greenest” value for that particular position.

XQuery specification of such processing is shown in Figure 3. We currently assume a simplified data representation, where the input data is simply a set of pixels. Each pixel stores the latitude, longitude, time, and 16-bit measurements for the 5 bands. The code iterates over the two-dimensional space for which the output is desired. Since the order in which the points are processed is not important, we use the directive *unordered*. Within an iteration of the nested for loop, the *let* statement is used to create a sequence of all pixels that correspond to the those spatial coordinates. The desired result involves finding the pixel with the best NDVI value. In XQuery, such reduction can only be computed recursively.

4 Compiler Analysis

In this section, we describe the various analysis, transformation, and code generation issues that are handled by our compiler. Initially, we summarize the challenges involved in compiling reductions specified in XQuery.

4.1 Overview of the Compilation Problem

For datasets that are stored as flat files, there are two ways in which XQuery is likely to be compiled. The first will be to translate it into an imperative language like C/C++, for which efficient compilers are available. The second will be to generate code for a middleware or runtime system, which offers data handling capabilities. Such systems also often offer interfaces based upon imperative languages. Thus, XQuery codes will likely need to be translated into codes in an imperative language.

Consider the code shown in Figure 3. Suppose, we translate it to an imperative language like C/C++, ignoring the *unordered* directive, and preserving the order of the computation otherwise. It is easy to see that the resulting code will be very inefficient, particularly when the datasets are large. This is primarily because of two reasons. First, each execution of the *let* expression will involve a complete scan over the dataset, since we need to find all data-elements that will belong to the sequence. Second, if this sequence involves n elements, then computing the result will require $n + 1$ recursive function calls, which again is very expensive.

We can significantly simplify the computation if we recognize that the computation in the recursive loop is a reduction operation involving associative and commutative operators only. This means that instead of creating a sequence and then applying the recursive function on it, we can initialize the output, process each element independently, and update the output using the identified associative and commutative operators. A direct benefit of it is that we can replace recursion by iteration, which reduces the overhead of function calls. However, a more significant advantage is that the iterations of the resulting loop can be executed in any order. Since such a loop is inside an *unordered* nested *for* loop, powerful restructuring transformations can be applied. Particularly, the code resulting after applying *data-centric* transformation [7, 9] will only require a single pass on the entire dataset.

Thus, the first three key compiler analysis and transformation tasks are: 1) recognizing that the recursive function involves a reduction computation with associative and commutative operations, 2) transforming such a recursive function into a *foreach loop*, i.e., a loop whose iterations can be executed in any order, and 3) restructuring the nested unordered loops to require only a single pass on the dataset. Note that the essential analysis required for the first and the second tasks is the same.

Translating XQuery to imperative languages also leads to a new challenge, which is *type inferencing*. We need to deduce the types of the data-structures used in the target languages. The type system used in XQuery is significantly different from that in popular imperative languages, which makes the type inferencing problem a non-trivial one.

4.2 Analysis of Recursive Functions

We now focus on analyzing recursive function with the following three goals: 1) identifying reduction computations involving associative and commutative operators, and 2) replacing them with a *foreach* loop.

Our analysis requires that the recursive function has the following canonical form:

```
define function F($t) {
  if (p1) then F1($t)
  else F2(F3($t), F4(F(F5($t))))
}
```

where, 1) The input $t must be a sequence, 2) F5 must be a subsequence of $t, including all but the first element of $t (i.e. subsequence($t,2) in XQuery), 3) Each of F1($t) and F3($t) must return constant values or should be functions of only the first element of the sequence $t.

Our algorithm is presented in Figure 11. The algorithm analyzes the abstract syntax tree (AST) of the function. Initially, it processes all leaf nodes of the tree. We focus on nodes of two types, the nodes that are a recursive call to the function F and the nodes that denote a variable defined by such a recursive call. In either of these cases, if the node is used as part of a return value, the node is inserted into the set S. Note that a recursive reduction function may compute multiple simple types, each through a different reduction operation. For example, an averaging function may compute a sum field and a count field, applying reduction operations *add* and *add by one*, respectively. Our algorithm separates the recursive calls used for computing each distinct field. Thus, the set S is partitioned into k subsets, $S_1, \ldots, S_k$.

Our algorithm subsequently processes each of these sets independently. For each node n in a set S_i, we apply the function $Findnode(n)$. This function finds the closest ancestor of n that has more than one child. The node thus obtained combines the value at n with another value. Next, we need to consider several different cases. If the set S_i has only one element, let t be the result of $Findnode$ to this element. If t denotes an associative and commutative operation, and if the function matches the canonical structure shown above, then we know that the recursive function performs a reduction computation. Here, we mark the sub-tree with t as the root as the reduction function. This function is used for transforming the recursive function to a *foreach* loop.

If the set S_i includes more than one element, we need to check for several conditions. We need to ensure that each node in S_i is in a different or *mutually exclusive* control path, i.e., invocation of the function results in at most one recursive call. If no language annotation is available for the compiler, we require that the results of $Findnode(n)$ for each n in S_i, i.e. the nodes in the set R, denote the same associative and commutative operation.

4.3 Data-Centric Execution

Replacing the recursive computation by a *foreach* loop is only an enabling transformation for our next step. The key transformation that provides a significant difference in the performance is the *data-centric transformation*, which is described in this section.

In Figure 12, part (a), we show the outline of the satellite data processing code after replacing recursion by iteration. Within the nested *for* loops, the *let* statement and the recursive function are replaced by two *foreach* loops. The first of these loops iterates over all elements in the document and creates a sequence. The second *foreach* loop performs the reduction by iterating over this sequence.

The code, as shown here, is very inefficient because of the need for iterating over the entire dataset a large number of times. If the dataset is disk-resident, it can mean extremely high overhead because of the disk latencies. Even if the dataset is memory resident, this code will have poor locality, and therefore, poor performance. A minor improvement to this code is shown in Figure 12, part (b). A loop fusion is performed to require only a single *foreach* loop.

Since the input dataset is never modified, it is clearly possible to execute such code to require only a single pass over the dataset. However, the challenge is to perform such transformation automatically. We apply the *data-centric* transformation that has previously been used for optimizing locality in scientific codes [7, 9]. The overall idea here to iterate over the available data elements, and then find and execute the iterations of the nested loop in which they are executed. As part of our compiler, we apply this transformation to the intermediate code we obtain after removing recursion and performing loop fusion. Because of the language constructs and the nature of the applications we focus on, the details of the algorithm we use are different from the previous work reporting such transformations.

We use the following approach. We create an abstract iteration space, corresponding to the nested *for* loops within the *unordered* construct. Then, for each element in any document that is accessed in the program, we determine a necessary and sufficient condition for the element to be read in an iteration i of the abstract iteration space. If a necessary and sufficient condition cannot be found, our compiler cannot perform the transformation. Otherwise, using this condition, we synthesize a mapping from an element to the set of iterations in which it is accessed. In our example, this is a singleton set.

An array of output elements corresponding to the iteration space is allocated and initialized in the beginning. Subsequently, each element from the document is accessed and mapped to zero or more iterations in which it can be accessed. Then, the computations associated with these iterations are performed, and the corresponding elements of the output array are modified. The outline of the resulting code for the satellite data processing is shown in Figure 12, part (c).

4.4 Type Analysis and Conversion

As stated previously, our compiler generates code for Active Data Repository (ADR) [3], which uses C++ virtual functions to specialize the processing for a particular application. To generate C++ code for a XQuery program, one challenging task is the correct and efficient conversion of various types in XQuery to corresponding C++ types. This problem needs to be addressed by any compiler for XQuery whose target code is in an imperative or object-oriented language.

XQuery is a statically typed functional language. The type annotations of each operand, argument, and function are always either explicitly declared or can be generated while validating against a Schema. This static feature of the type system provides various possibilities for compiler optimizations and analysis, such as the well studied method for static type checking [2], and in our case, static type analysis for translation to an imperative language.

Although type systems of both C++ and XQuery are static, the mapping between these two is not straightforward. There are several issues that make the translation problem challenging.

```
typeswitch ($pixel) {
   case element double_pixel
      return max( $pixel/d_vaule,0)
   case element integer_pixel
      return max($pixel/i_value,0)
   default 0
}
```

Fig. 4. XQuery expression with different types

First, unlike variables in an imperative language, an expression in XQuery may be associated with values of several different types. An example of such an expression is shown in Figure 4. The expression here may return either a double or an integer, depending upon the type of the variable $pixel$. To perform the translation, our compiler needs to collect the static types of all possible branches and compute a union of these types. For union of simple types, we will generate a corresponding union type in the target code. For union of complex types, we use the polymorphism of C++ language.

The second issue is related to the parametric polymorphism of XQuery. An actual argument of a function is only required to be of a subtype of the declared type of the corresponding formal function parameter. For example, the max function in Figure 4 can be invoked with an argument whose type may be either integer or double. Moreover, some parameters may be declared as $AnyType$, implying that their type can only be known by validating against an XML Schema. Therefore, to infer the type of a formal parameter, we need to gather information about the actual arguments from call sites, and if necessary, from Schema definitions. In cases where such parametric polymorphism is used, we use *function cloning*, i.e., we generate a copy of the function for each distinct type we infer. We believe this gives better performance than using polymorphism of C++, considering the limited occurrences of such functions in the actual cases we studied.

Overall, our static type analysis algorithm is based on constraint-based type inference algorithm [8], and the static type checking algorithm listed as part of the specification of XQuery formal semantics [6]. The goal of this algorithm is to infer the collection of all possible static types for a given expression, which will be used to guide the code generation in our compiler. The algorithm is top-down and recursive, since the static type of an expression depends only on the types of its sub-expressions.

The first step in our algorithm is to initialize the type of each expression, if the type is explicitly defined in the XQuery program or in an XML Schema. After initialization of type variables, we will analyze possible types for an expression by propagating type variables from subexpressions according to possible run-time data flow. Specially, if the target expression has only one possible value, the outcome is just propagated from its operands. For operands with different but compatible types, a *least upper bound* for

```
TypeAnalysis (Expression $E$ ) {
  Set $S = \phi$
  if type $t$ is explicitly assigned to $E$
    return $t$ ;
  else if $E$ is a function call expression {
    $E_1 =$ body of function definition expression of $E$
    for each actual argument $e^i$ of $E$
      $TypeAnalysis$ ( $e^i$ )
    assign type of actual arguments to formal parameters
    return $TypeAnalysis($ $E_1$ )
  }
  else if  $E$ has single control flow path {
    for each subexpression $E^i$ $of$ E {
      S=S   $\{TypeAnalysis($ $E^i$ $)\}$
    return  least upper bound of $S$
  }
  else for each control path $P^i$ $of$ E {
    S=S   $\{TypeAnalysis($ $P^i$ $)\}$
    return  $S$
  }
}
```

Fig. 5. Algorithm for Static Type Analysis

these compatible types is returned as the result. If the target expression is a function call expression, information about the types of the actual arguments needs to be gathered from the call sites. This is done by applying the algorithm for each actual argument. The resulted types are assigned to each formal parameter, and the function body of the corresponding function definition is processed by the algorithm. For any other expression whose value is defined by subexpressions in more than one branch, we simply apply the algorithm for all sub-expressions in each branch and compute a union for all possible results. The detailed algorithm is listed in Figure 5.

After finishing type analysis for each expression in a XQuery program, we can generate C++ code that correctly implements the type system of XQuery. The C++ code generated after type analysis and code generation for the code in Figure 4 is shown in Figure 6. Because the $typeswitch$ expression may return either a double or an integer type, a $union$ type is declared to keep the result of this expression. The variable $\$pixel$ may be bound to two complex types when validated against a Schema. Therefore, we declare a superclass for $\$pixel$, from which two subclasses can be derived. Also in this example, because the actual arguments of the function max can be either double or integer, a clone of the function is generated for each type.

5 Experimental Results

To evaluate our techniques and current prototype compiler implementation, we have evaluated the impact of our restructuring transformations, i.e. removing recursion and applying data-centric execution, on sequential execution.

```
class t_pixel pixel;
...
...
struct tmp_result_1 {
    union {
        double r1 ;
        int r2 ;
    };
};
...
tmp_result_1 t1;
if (pixel.tag_double_pixel)
    t1.r1 = max_1(pixel.d_vaule,0) ;
else if (piexl.tag_integer_pixel)
    t1.r2 = max_2(pixel.i_value,0) ;
else
    t1.r2 = 0;
```

Fig. 6. Compiler Generated C++ code

We compared sequential performance between two versions, `opt` and `naive`. The `naive` version corresponds to a direct translation of XQuery codes to C++. The `opt` version includes replacing recursion with iteration and application of data-centric transformation. The resulting code is similar to the one shown in Figure 12, part (c).

We used four applications for performing the comparison. `sum` is a very simple sum reduction, where values associated with pixels having the same x and y coordinates are added together. `irreg` is a simple irregular reduction, where the values associated with edges are used to increment the values associated with corresponding nodes. `satellite` and `mg-vscope` are two real applications we described in Section 3.

Our experiments were conducted on a 933 MHz Pentium III workstation, with 256 MB of RAM, and running linux version 7.1. We used four synthetic datasets for each

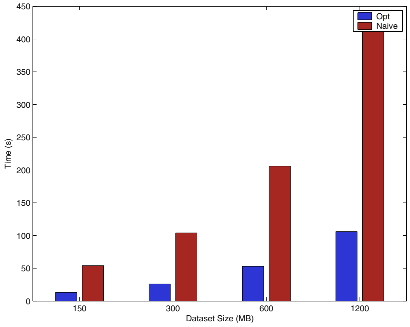


Fig. 7. Evaluating the Benefit from Transformations, `sum` kernel

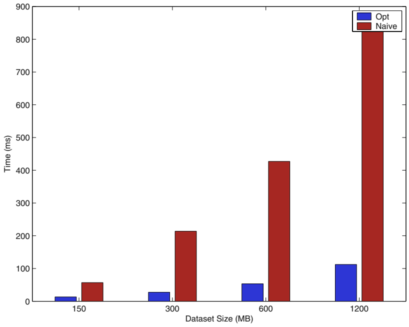


Fig. 8. Evaluating the Benefit from Transformations, `irreg` kernel

of four applications. Their approximate sizes were 150 MB, 300 MB, 600 MB, and 1.2 GB, respectively. Thus, we studied the benefits of transformations as the size of the dataset is increased.

The results from `sum` are presented in Figure 7. The output produced by this kernel is a 2 × 2 array. Thus, the `naive` version requires 4 passes over the entire data, whereas, the `opt` version requires only a single pass. The difference between the `opt` and `naive` versions is consistently a factor of 4. Thus, it appears that the execution time is dominated by memory and disk access times.

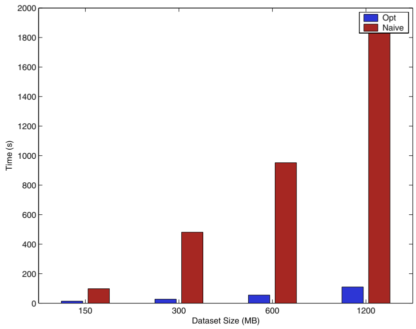


Fig. 9. Evaluating the Benefit from Transformations, `satellite` application

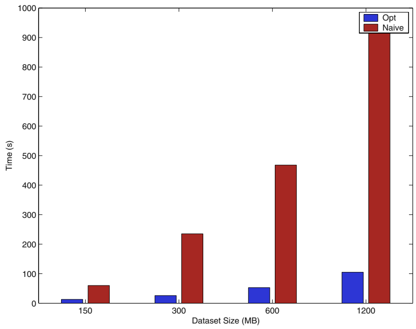


Fig. 10. Evaluating the Benefit from Transformations, `mg-vscope` application

The results from `irreg` kernel are presented in Figure 8. The output of this application is a 12 × 12 array. The difference between the two versions is a factor of 4 for the 150 MB dataset, and a factor of 8 for each of the other three datasets. Note that the first dataset can be cached on memory, therefore, subsequent passes over the data only require memory accesses. In comparison, the other three dataset cannot be cached and multiple passes over the dataset result in more disk accesses. This accounts for higher difference in performance for the larger datasets.

The results from `satellite` and `mg-vscope` are presented in Figures 9 and 10, respectively. The output of `satellite` kernel is a 6 × 3 array and the output of `mg-vscope` kernel is a 3 × 3 array. For `satellite`, the difference in performance between `opt` and `naive` versions is a factor of 7 for the 150 MB dataset, and between 16 and 18 for other datasets. This shows that when the data cannot be cached in main memory, the execution time is dominated by disk access time. Thus, increasing the number of passes on data result in almost linear increase in execution times.

The results from `mg-vscope` are very similar. The difference in performance is a factor of 4.5 for the 150 MB dataset, and then, nearly a factor of 9 for other datasets.

6 Related Work

Only a limited amount of work currently exists on optimizing XQuery programs. Park *et al.* have focused on processing of recursive XML records [11]. Choi, Fernandez,

```
Analyze(AST $T$) {
   $S = \phi$
   for each leaf node $n$ in $T$
      if $n$ is a recursive function call node
         if $n$ is used as part of a returned value
            $S = S \cup \{n\}$
      else if $n$ is a variable defined by a recursive call
         if $n$ is used as part of a returned value
            $S = S \cup \{n\}$

   Partition nodes in $S$ based upon the field computed
   let $S_1, \ldots, S_k$ denote the $k$ partitions

   for $i = 1, \ldots k$
      $R = \phi$
      for each node $n$ in $S_i$
         $R = R \cup \{Findnode(n)\}$
      if $S_i$ and $R$ are singleton sets
         let $R = \{t\}$
         if $t$ is an associative and commutative operation
            mark the sub-tree with $t$ as the root as the reduction operation
         else mark the function as non-transformable
      else
         if each node in $S_i$ is in different control path and
               each node in $R$ is the same associative and commutative operation
            let $t'$ be the least common ancestor of all nodes in $R$
            mark the sub-tree with $t'$ as the root as the reduction operation
         else mark the function as non-transformable
}

Findnode(node $n$) {
   let $p$ = parent of $n$
   if $p$ has at least two children including $n$
      return $p$
   else return Findnode($p$)
}
```

Fig. 11. Algorithm for Analysis of Recursive Functions

and Simeon have been developing formal semantics of XQuery, as the basis for implementation and optimization [4]. DeHaan *et al.* have developed techniques and a system for translating from XQuery to SQL [5]. We are not aware of any existing work on optimizing XQuery for flat-file datasets, or processing of generalized reductions using XQuery.

Lieuwen and Dewitt have used compile-time loop transformations to optimize the performance of join operations [10]. However, the structure of join operations is very different from reduction computations. In the database community, optimization of ag-

```
unordered(
   for $i in ($minx to $maxx)
      for $j in ($miny to $maxy)
         foreach element $e in document(”satellite.xml”)/data/pixal
            if (( $e/x = $i) and ($e/x = $j ))
               Insert $e to the sequence $p
         Initialize the output
         foreach element $e in $p
            Apply the reduction function and update output
         return output
)
```

(a)

```
unordered(
   for $i in ($minx to $maxx)
      for $j in ($miny to $maxy)
         Initialize the output
         foreach element $e in document(”satellite.xml”)/data/pixal
            if (( $e/x = $i) and ($e/x = $j ))
               Apply the reduction function and update output
         return output

)
```

(b)

```
   for $i in ($minx to $maxx)
      for $j in ($miny to $maxy)
         Initialize output[i,j]
   foreach element $e in document(”satellite.xml”)/data/pixal
      $i = $e/x
      $j = $e/y
      if ($i $\geq$ $minx) and ($i $\leq$ $maxx) and
            ($j $\geq$ $miny) and ($j $\leq$ $maxy)
         Apply the reduction function and update output[i,j]
```

(c)

Fig. 12. Transformations on the Satellite Data Processing Code: Removing Recursion (a), Applying Loop Fusion (b), and Data-Centric Transformation (c)

gregation and group-by queries has been studied by many researchers. For example, Shatdal has studied SMP parallelization of aggregation queries [12]. However, the reductions involved in our target applications are significantly more complex, and cannot be expressed or processing efficiently using SQL-like languages. Our contribution in this paper is demonstrating how XQuery can be used for expressing them, and aggressive transformations can be applied for optimizing them.

7 Conclusions

With the wide acceptance of XML as the format for exchanging information, we envision that XML query language XQuery will be a popular language for specifying processing over datasets. This paper has described our initial efforts in compiling XQuery. Particularly, our focus has been on data-intensive reductions over scientific datasets.

Our work has demonstrated that XQuery simplifies the specification of processing over datasets. At the same time, however, naive translation of XQuery results in very slow execution. We have shown that aggressive restructuring transformations are the key to improving performance, and their application results in several-folds improvement in performance.

References

1. Asmara Afework, Michael D. Beynon, Fabian Bustamante, Angelo Demarzo, Renato Ferreira, Robert Miller, Mark Silberman, Joel Saltz, Alan Sussman, and Hubert Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.
2. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. W3C Working Draft, available from http://www.w3.org/TR/xquery/, November 2002.
3. Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman, and Joel Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 1997 International Conference on Data Engineering*, pages 375–384. IEEE Computer Society Press, April 1997.
4. Byron Choi, Mary Fernandez, and Jerome Simeon. The XQuery Formal Semantics: A Foundation for Implementation and Opitmization. May 2002.
5. David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Ozsu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Coding. In *Proceedings of the ACM SIGMOD*. ACM Press, June 2003.
6. D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simion, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft, available from http://www.w3.org/TR/query-semantics/, November 2002.
7. Renato Ferreira, Gagan Agrawal, and Joel Saltz. Compiler supported high-level abstractions for sparse disk-resident datasets. In *Proceedings of the International Conference on Supercomputing (ICS)*, June 2002.
8. Palsberg. J and M. Schwartzbach. Object-Oriented Type Inference. In *ACM SIGPLAN Sixth Annual Conference on Obejct-Oriented Programming Systems, Languages and Applications*, 1991.
9. Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 346–357, June 1997.
10. Daniel F. Lieuwen and David J. Dewitt. A Transformation Based Approach for Optimizing Loops in Database Programming Languages. In *Proceedings of ACM SIGMOD*, pages 91–100, 1992.
11. Chang-Won Park, Jun-Ki Min, and Chin-Wan Chung. Structural Function Inlining Techniques for Structurally Recursive XML Queries. In *Proceedings of Conference on Very Large Databases (VLDB)*, September 2002.

12. Ambuj Shatdal. Architectural considerations for parallel query evaluation algorithms. Technical Report CS-TR-1996-1321, University of Wisconsin, 199.
13. B. B. Yao, M. T. Ozsu, and J. Kennleyside. XBench – A Family of Benchmarks for XML DBMSs. In *Proceedings of EEXTT 2002 and DiWeb 2002, Lecture Notes in Computer Science Volume 2590*, 2002.

A Query Algebra for Fragmented XML Stream Data

Sujoe Bose, Leonidas Fegaras, David Levine, and Vamsi Chaluvadi

Department of Computer Science and Engineering
University of Texas at Arlington
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015, USA
bose@cse.uta.edu
http://www.cse.uta.edu

Abstract. The increased usage of mobile devices coupled with an unprecedented demand for information has pushed the scalability problem of pull-based data service to the focus. A broadcast model of streaming data over a wireless medium has been proposed as a viable alternative for information dissemination. In the streaming broadcast model, servers broadcast data in an asynchronous and unacknowledged mode while clients process personalized and complex queries locally, relieving the load on the server. We address the query processing of streamed XML data, which is fragmented into manageable chunks for easier synchronization. Although there has been some work done in defining algebras that model XQueries on XML documents, no work has been done in defining query algebras for fragmented XML stream data. We define a model for processing fragmented XML stream data, using the concept of *holes* and *fillers*. This model offers the flexibility required by the server to disseminate data in manageable fragments, whenever they become available, and to send repetitions, replacements and removal of fragments. We then present a query algebra for XQuery that operates on this streamed XML data model. The XML fragments are operated upon in a continuous pipelined fashion without the need of materializing the transmitted document at the client site.

1 Introduction

1.1 Motivation

Traditionally, client-server architectures are service oriented, in which the servers process client requests in the form of queries over the server database and return the data pertaining to the client requests. In this setup, the onus is on the server to process the requests at the behest of the clients and to send back the results. The dramatic increase in the mobile devices and their data-providing applications, coupled with the complex and customized queries from the clients, may push the load on the servers to prohibitive proportions. The alternative

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 195–215, 2004.

method is to disseminate the data available at the servers (both static and real-time data) in a broadcast (or multicast) medium, with multiple channels of communication, on which the clients listen.

We envision a large class of applications for a push-based data model we call broadcast of streaming XML data, or *XStreamCast*. We assume that a typical configuration will consist of a small number of servers that transmit XML data over high-bandwidth streams and many clients that receive this data. Stream data may be infinite or repeated and transmitted in a continuous stream, such as measurement or sensor data transmitted by a real-time monitoring system continuously. Even though our model resembles a radio transmitter that broadcasts to numerous small radio receivers, our clients can tune-in to multiple servers at the same time to correlate their stream data. In contrast to servers, which can be unsophisticated, clients must have enough memory capacity and processing power to evaluate sophisticated, continuous queries on the XML data stream. For example, a server may broadcast stock prices and a client may evaluate a continuous query on a wireless, mobile device that checks and warns (by activating a trigger) on rapid changes in selected stock prices within a time period.

We believe that there will be a growing interest in query processing of continuous data streams as the network bandwidth increases at rates higher than disk access times, which makes it inefficient to cache stream data on disk and process it using conventional database techniques. Moreover, pushing data to multiple clients is highly desirable when a large number of similar queries are submitted to a server and the query results are large, such as requesting a region from a geographical database. By distributing processing to clients, we reduce the server workload while increasing its availability. More importantly, when the push-based model is used, a client can correlate data between two or more sources by tuning-in and joining the data streams on-line, while, when the pull-based model is used, a client must submit one query to each server that holds the data and then join the results using a combining query. There are many sophisticated distributed query processing techniques that address the latter, such as semijoins, but, in general, large amounts of data must be transmitted from servers to clients before they are combined.

The standardization of XML, along with its widespread adoption and usage, presents it as the language of choice for communications between cooperating systems. XML, along with a rich set of supporting specifications and standards, has been the focus of current research. Even though there are several proposals for XML query languages [13], we adopt the XQuery standard proposed by World-Wide Web Consortium (W3C) [6]. While the advantages of continuous processing of streamed data compared to computation on stored data have been understood and realized [2, 3], not much work has been done in the streamed XML front with XQueries as the stream processing function. Much of the work related to XML stream processing has been focused on XML filtering, i.e., matching a large number of client XPath queries with XML documents at the server side and multicasting the matched documents to the appropriate clients [15, 1]. Although there have been some efforts on defining algebras that model XQueries on XML documents, to our knowledge, no work has been done in defining query algebras for streamed XML data.

As an example consider two XML streams: the *commodities* stream that lists the commodities by vendor along with its prices in USD:

```
<commodities>
   <vendor>
      <name> Wal-Mart </name>
        <items>
           <item>
              <name> PDA </name>
              <make> HP </make>
              <model> PalmPilot </model>
              <price currency="USD">315.25<price>
           </item> ...  </items>
   </vendor>
   ...
</commodities>
```

and the *currency* stream that provides currency conversion information:

```
<currencies>
   <currency>
      <code> USD </code>
         <name> US Dollars </name>
         <rates>
            <currency>
               <code> GBP </code>
               <buying> 1.34 </buying>
               <selling> 1.32 </selling>
            </currency> ... </rates>
   </currency>
   ...
</currencies>
```

We would like to issue a query to get a list of prices for the PalmPilots in GBP sorted by price, lowest to highest. The XQuery would look like:

```
<result>
{ for $c in stream(commodities)//vendor/items/item
  where $c/model/text() = "Palm-Pilot"
  return
    <vendor> $c/../../name/text() </vendor>
    <price>
      for $u in stream(currency)//rates/currency
      where $u/code/text() = "GBP"
      and $u/../../code/text()=$c/price/@currency
      return $u/buying/value()*$c/price/value()
    </price>
  sortby($c/price)
} </result>
```

The challenge is in evaluating this query against the XML data from the two streams. The implicit join between the commodities and the currency streams in the above query requires a state of unbounded length, since the GBP currency information may come at the end of the currency stream. Moreover, the sortby clause makes the query a blocking query, since no result can be obtained without reading both streams.

1.2 Our Approach

Since an XML stream can be visualized as an infinite sequence of data items belonging to some data set, in our framework, we fragment the stream into manageable chunks of information. These chunks, or fragments, are related to each other and can be reassembled at the client side upon arrival. However, the focus of our work is not in reconstructing the stream in the client devices, but rather to evaluate ad-hoc queries against these fragments at the client side in real time and produce results. Our XML fragments follow the *Hole-Filler model* in which every fragment is treated as a "filler" and is associated with a unique ID. When a fragment needs to refer to another fragment in a parent-child relationship, it includes a "hole", whose ID matches the filler ID of the referenced filler fragment. Note that a document can be fragmented to produce fillers and holes in arbitrary ways. Furthermore, the fragments can be arranged to any depth and breadth in a document tree.

Fragmenting a stream has the following advantages. While it is hard to synchronize on infinitely long streamed XML data, it is relatively easy to synchronize on fragmented XML data. The fragmentation also is necessary when pieces of data are made available to the server in real-time, so that information that is available may be transmitted as and when is received. For example, in a stock quote stream, the enumerated set of ticker symbols are near-static, however the stock quotes on those symbols change, and these changes need to be transmitted often as fragments, wrapped with sufficient context information. The stream fragments can be repeated for redundancy and to accommodate client devices that connect sporadically.

The novelty of this paper is in characterizing XML streams by a list of fragments, suitable for dissemination by servers and for processing by clients. This is captured in our streamed XML model by discrete fragments of XML data interrelated and intermittent in arrival, wherein a particular ordering of fragments is not guaranteed. This characterization allows servers to be able to repeat or replace fragments, introduce new ones or delete outdated ones. More importantly, this approach closely reflects the dynamics of data, especially in real-time applications, and provides the required controls on the server to make sure clients are synchronized with the data at the server. Another contribution of this paper is a novel query algebra based on the streamed XML model. This algebra commensurates with the main theme of our work: to directly evaluate the queries input by the client on streams of XML data without materializing it in memory. We extend our earlier work on an algebra for stored XML data [11] to operate on our streamed XML model rather than on stored XML documents. The challenge is in evaluating the operators against our streamed XML model, which consists of discrete hole and filler fragments interrelated to each other. Since, fragments are not guaranteed to arrive at a particular order, we need to take care of arrivals of filler fragments before the presence of a hole for the filler and vice versa. This leads to an algebra in which the evaluation on certain fragments may be suspended if the fragments have holes for which the fillers have not arrived and the path evaluation used in an algebraic operator leads to the hole.

1.3 Assumptions

In our push-based broadcasting framework, we make the following assumptions about the environment and the characteristics of the components involved. We assume that client devices connect into multiple data streams asynchronously and have reasonable amount of resources in terms of computing and storage. The communication medium is assumed to be noisy and lossy, but we will not address losses in the communication medium and handling a lossy medium is considered as future work. However, the fact that the stream fragments can be repeated, replaced, removed or added by the server, offsets the problem of the communication losses. It also enables clients to connect into the stream and be able to process the data stream. The server may send the fragments in any order. Another assumption is that data is well-typed. The server will transmit metadata periodically to provide the structure of the XML data being transmitted (described in Section 2.1). The queries are assumed to be input through a menu interface or manually. The queries can be generic at first and can be refined to return desired results. Since clients can tune-in to streams at different points in time, and the servers disseminate fragments of data asynchronously, the results of queries will reflect the time in which the client are connected to the streams as well as the duration of the connections.

1.4 Layout of the Paper

In Section 2, we present a formal model for the streamed XML data, based on the Hole-Filler model, which serves as the basis for our query algebra. In Section 3, we review our earlier work on an algebra for stored XML data and, in Section 4, we extend this algebra to process streams of fragmented XML data. Our streamed query algebra operates on the fragments to produce continuous results in a pipelined fashion. Finally, we present the related work in Section 5 and we conclude in Section 6.

2 Our Model for Streamed XML Data

2.1 The Fragmented Hole-Filler Model

In our model, XML documents are transmitted in fragments. These fragments may be disseminated by a server in any order. To be able to relate fragments with each other, we introduce the concept of *holes* and *fillers*. A hole represents an empty node into which another rooted subtree, called a filler, could be positioned to complete the tree. The filler can in turn have holes in it, which will be filled by other fillers. Unique IDs are assigned to the holes. The fragment that corresponds to a hole has the same ID in its header. Thus, by substituting holes with the corresponding fillers, we can reconstruct the whole XML tree at the client side as it was in the server side. However, reconstructing the original XML tree is not always a good idea, since clients would have to wait for the end of the stream to begin processing. As will be discussed in the next section, our goal is to apply our query algebra on this streamed model and to process XML fragments as they become available.

At the server side, the XML document may be fragmented by recursively pruning the tree, inserting a hole at every point a tree is pruned and associating it with an ID. The server may prune the tree in an arbitrary way. This arrangement, of associating holes with fillers, takes care of out-of-sequence transmission, repetitions, replacements and removals. Another important piece of information transmitted by the server, is the *Tag Structure* of the document transmitted in the stream as a separate fragment. This tag structure is a structural summary that provides the structural make-up of the XML data and captures all the valid paths in the data. This information is useful while expanding wildcard path selections in the client queries. Moreover, this structure gives us the convenience of abbreviating the tag names with IDs (not used here) for compressing stream data. The following is a tag structure corresponding to the commodities stream:

```
<stream:structure>
   <tag name="commmodities" id="1">
       <tag name="vendor" id="2">
           <tag name="name" id="3"/>
           <tag name="items" id="4">
               <tag name="item" id="5">
                   <tag name="name" id="6"/>
                   <tag name="make" id="7"/>
                   <tag name="model" id="8"/>
                   <tag name="price" id="9"/>
               </tag>
           </tag>
       </tag>
   </tag>
</stream:structure>
```

As an example of the hole-filler model of fragmenting XML data, consider the commodities XML document stream with thousands of vendors, where each vendor potentially provides thousands of items. This document, in its entirety, may be too large to send as a single document. Moreover, data corresponding to all the vendors may not be available at the same time. The commodities stream would be transmitted in the following fragments, where each fragment is a filler for a hole in some other filler fragment:

Fragment 1:

```
<commodities>
   <vendor>
      <name> Wal-Mart </name>
        <items>
          <stream:hole id="10" tsid="5"/>
          <stream:hole id="20" tsid="5"/>
          ...
   </vendor> ... </commodities>
```

Fragment 2:

```
<stream:filler id="10" tsid="5">
  <item>
    <name> PDA </name>
    <make> HP </make>
    <model> PalmPilot </model>
    <price currency="USD">315.25<price>
  </item></stream:filler>
```

Fragment 3:

```
<stream:filler id="20" tsid="5">
  <item>
    <name> Calculator </name>
    <make> Casio </make>
    <model> FX-100 </model>
    <price currency="USD">50.25<price>
  </item></stream:filler>
```

Fillers are associated to holes by matching filler IDs with hole IDs. The filler with $id = 0$ is always the root filler, and hence the root of the fragmented document. The *tsid* attribute identifies the ID of the tag structure element corresponding to the filler fragment element. Using the tag structure ID, the structural context of the fragment can be deduced by looking-up the tag structure sent by the server intermittently. The filler of fragment 2 fills the hole with $id = 10$ inside fragment 1. It can be seen that fragment 1 is near-static while fragments 2 and 3 are not, due to variations in price. While fragment 1 can be transmitted earlier, the fragments 2 and 3 can be transmitted as soon as the price on an item is received. Note that the fragmentation can be done at any level in the XML tree, based on the flexibility in the granularity of fragmentation in various data domains. In the example above, we could have chosen to further fragment the right filler such that the "price" element, which is dynamic, be transmitted as a separate filler fragment.

2.2 Formal Definition of the Model

In this section, we formalize the fragmented stream model of XML data outlined in Section 2.1. The basic stream components transmitted by the server are fillers and holes, each with its own ID. In order for the server to be able to repeat, replace and remove fragments, the basic stream constructs are extended with repeat, replace and remove primitives. The server can also send an end-of-stream fragment, which is a marker for end of transmission. The stream content (γ) can be described by the following grammar:

$$\begin{array}{rl}
\gamma ::= & \text{<filler id=“m” tsid=“n”>}x\text{ </filler>} \\
| & \text{<repeat id=“m” tsid=“n”>}x\text{ </repeat>} \\
| & \text{<replace id=“m” tsid=“n”>}x\text{ </replace>} \\
| & \text{<remove id=“m” tsid=“n”/>} \\
| & \text{<eos/>} \\
| & \gamma \ ; \ \gamma
\end{array}$$

where m and n are numbers, representing the *id* and *tsid* attributes respectively, and x is a valid XML fragment, defined as follows (XML attributes are ignored):

$$\begin{array}{rl}
x ::= & \text{<tag>}x\text{ </tag>} \\
| & \text{<tag/>} \\
| & \text{<hole id=“m” tsid=“n”/>} \\
| & \text{PCDATA} \\
| & x \ x
\end{array}$$

The root of the document is always transmitted with filler id = “0”. For convenience, we represent the stream components with the following syntactic shorthands, where m and n stand for the id and tsid attribute values respectively in the original form:

$$\begin{array}{rll}
\gamma ::= & F(m,n,x) & \text{(a \textit{filler})} \\
| & R(m,n,x) & \text{(a \textit{repeat})} \\
| & U(m,n,x) & \text{(a \textit{replace})} \\
| & D(m,n) & \text{(a \textit{remove})} \\
| & eos & \text{(an \textit{end of stream})} \\
| & \gamma \ ; \ \gamma & \text{(a \textit{stream sequence})}
\end{array}$$

The stream components thus contain XML fragments of the original document and these XML fragments in turn may contain holes, which will be filled by other stream components. Note that we do not model silence during transmission, which is inherent in stream-based models. The reason is that we do not want to clutter the grammar and algebra with entities not contributing to the processing. We do however consider this aspect when evaluating certain operators that read from multiple streams, such as joins, since the streams may not be synchronized, or worse, one of the streams may be blocked.

2.3 Reconstruction of the XML Document from Stream Fragments

The fragmented XML document in the form of holes and fillers can be reconstructed, in its entirety, at the client by recursively filling the holes with the corresponding fillers. We define a stream transformation function $\mathcal{T}$, which maps XML fragments represented by γ onto an XML document. The stream transformation function involves the process of maintaining a mapping between filler IDs and the fillers, based on the stream components, and recursively filling holes with fillers with corresponding IDs. $\mathcal{T}$ can then be defined in terms of the filler mapping function $\mathcal{T}_1$, and the hole substitution function $\mathcal{T}_2$. To accurately depict the causal semantics of the stream primitives, which modify the mapping between filler IDs and the fillers, we represent function $\mathcal{T}_1$ as taking the stream as its argument, and returning an environment containing the mapping between the filler IDs and fillers. The hole substitution function $\mathcal{T}_2$ then substitutes the holes in the fragments recursively. Note that this function takes care of out-of-order arrival of holes and their corresponding fillers. After the holes are substituted, we have a set of XML fragments each representing various possible sub-trees, devoid of holes, of the original XML document. To retrieve the original document from this set, we retrieve the XML fragment with $id = 0$.

$\mathcal{T}$ is defined on a stream γ as $\mathcal{T}(\gamma) = \mathcal{T}_2(\mathcal{T}_1(\gamma, \emptyset))[0]$, where $\mathcal{T}_1$ returns an environment ζ, which is a set of bindings that, for each fragment, it binds the fragment ID to the fragment content. The filler mapping function, $\mathcal{T}$, is defined over the stream γ, and is shown in a case method of function definitions, over the possible stream primitives as follows:

$$\begin{aligned}
\mathcal{T}_1(F(m,n,x),\zeta) &= \zeta \cup \{(m,x)\} \\
\mathcal{T}_1(R(m,n,x),\zeta) &= \zeta \cup \{(m,x)\} \\
\mathcal{T}_1(D(m,n),\zeta) &= \{\,(k,y) \mid (k,y) \in \zeta,\ k \neq m\,\} \\
\mathcal{T}_1(U(m,n,x),\zeta) &= \mathcal{T}_1(D(m,n),\zeta) \cup \{(m,x)\} \\
\mathcal{T}_1(eos,\zeta) &= \zeta \\
\mathcal{T}_1(\gamma_1\,;\,\gamma_2,\zeta) &= \mathcal{T}_1(\gamma_2,\mathcal{T}_1(\gamma_1,\zeta))
\end{aligned}$$

Filler fragments and repeat fragments are appended into the environment ζ, while a delete fragment removes the mapping of an ID. A replace fragment causes the deletion of an existing filler with matching ID and the subsequent addition to the environment. A stream sequence is handled recursively and the end-of-stream fragment simply returns the final environment. The hole substitution function, $\mathcal{T}_2(\zeta)$ merges the fragments in ζ by filling holes with fillers:

$$\mathcal{T}_2(\zeta) = \{\,(k, y[m/x]) \mid (m,x) \in \zeta,\ (k,y) \in \zeta, \text{<hole id="m".../>} \in y\,\}$$

where $y[m/x]$ replaces <hole id="m".../> in y with x. Finally, after applying $\mathcal{T}_2$ to ζ, we retrieve the entire XML tree by indexing the root (of id="0").

Thus, given a query $q(x_1, \ldots, x_n)$ over n XML trees $x_1, \ldots, x_n$, and a query $q'(\gamma_1, \ldots, \gamma_n)$ over n streams of XML data $\gamma_1, \ldots, \gamma_n$, we say that q and q' are equivalent iff:

$$\forall \gamma_i : \mathcal{T}(q'(\gamma_1, \ldots, \gamma_n)) = q(\mathcal{T}(\gamma_1), \ldots, \mathcal{T}(\gamma_n))$$

that is, q and q' must return the same result for any input. We will use this natural transformation for proving the equivalence of our streamed XML algebra

$$
\begin{aligned}
\llbracket \rho_v(T) \rrbracket_\delta &= \{ < v = T > \} \\
\llbracket \sigma_{pred}(X) \rrbracket_\delta &= \{ t \mid t \in \llbracket X \rrbracket_\delta ,\ \llbracket pred \rrbracket_{\delta \circ t} \} \\
\llbracket \pi_{v_1,\ldots,v_n}(X) \rrbracket_\delta &= \{ < v_1 = t.v_1, \ldots, v_n = t.v_n > \ \mid t \in \llbracket X \rrbracket_\delta \} \\
\llbracket X \bigcup Y \rrbracket_\delta &= \llbracket X \rrbracket_\delta +\!\!+ \llbracket Y \rrbracket_\delta \\
\llbracket X \bowtie_{pred} Y \rrbracket_\delta &= \{ t_x \circ t_y \mid t_x \in \llbracket X \rrbracket_\delta ,\ t_y \in \llbracket Y \rrbracket_\delta ,\ \llbracket pred \rrbracket_{\delta \circ t_x \circ t_y} \} \\
\llbracket \mu^{v,path}_{pred}(X) \rrbracket_\delta &= \{ t \circ < v = w > \mid t \in \llbracket X \rrbracket_\delta ,\ w \in \mathcal{P}(\delta \circ t, path), \\
&\qquad\qquad \llbracket pred \rrbracket_{\delta \circ t \circ < v = w >} \} \\
\llbracket \Delta^{\oplus,head}_{pred}(X) \rrbracket_\delta &= \oplus / \{ \llbracket head \rrbracket_{\delta \circ t} \mid t \in \llbracket X \rrbracket_\delta ,\ \llbracket pred \rrbracket_{\delta \circ t} \} \\
\llbracket \Gamma^{v,\oplus,head}_{group,pred}(X) \rrbracket_\delta &= \{ \llbracket group \rrbracket_{\delta \circ t_1} \circ < v = \oplus / \{ \llbracket head \rrbracket_{\delta \circ t_2} \mid t_2 \in \llbracket X \rrbracket_\delta ,\ \llbracket pred \rrbracket_{\delta \circ t_2} , \\
&\qquad\qquad \llbracket group \rrbracket_{\delta \circ t_2} = \llbracket group \rrbracket_{\delta \circ t_1} \} > \\
&\qquad \mid t_1 \in \llbracket X \rrbracket_\delta \}
\end{aligned}
$$

Fig. 1. An Algebra for Stored XML Data.

to our algebra for stored XML data. Our effort concentrates on designing q', operating on the fragments, such that it will yield identical result as q operating on entire XML documents. We present this in a two step process: first, we give the semantics of our algebra q, for stored XML data (XML trees). We then transform this algebra into q', which operates on the fragmented XML model and produces identical results as q. Reconstructing the original XML tree is not always a good idea, since clients would have to wait for the end of the stream to begin processing. The emphasis of our work is in providing the semantics of a query algebra such that, executing the query on the fragments and then constructing the resulting document to form the final output produces the same result as constructing the entire document and then executing the query. Thus the results can be pipelined and produced as and when they occur.

3 An Algebra for Stored XML Data

In this section, we present our algebra for stored XML data introduced in earlier work [11]. Our streamed XML algebra, described in the next section, has the same algebraic operators but different data domains (streams rather than trees) and different semantics.

The algebraic bulk operators along with their semantics are given in Figure 1. The inputs and output of each operator are sequences of tuples that contain XML subtrees. These sequences are captured as lists of records and can be concatenated with list append, $+\!\!+$. There are other non-bulk operators, such as boolean comparisons, which are not listed here. The semantics is given in terms of record concatenation, $\circ$, and list comprehensions, $\{ \ e \mid \ldots \ \}$, which, unlike the set former notation, preserve the order and multiplicity of elements. The form

$\oplus/\{\ \cdots\ \}$ reduces the elements resulted from the list comprehension using the associative binary operator, $\oplus$ (a monoid, such as $\cup$, $+$, $*$, $\wedge$, $\vee$, etc). That is, for a non-bulk monoid $\oplus$, such as $+$, we have $+/\{a_1, a_2 \ldots, a_n\} = a_1 + a_2 + \cdots + a_n$, while for a bulk monoid, such as $\cup$, we have $\cup/\{a_1, a_2 \ldots, a_n\} = \{a_1\} \cup \{a_2\} \cup \cdots \cup \{a_n\}$.

The environment, δ, is the current record under consideration, and is used in the nested queries. Nested queries are mapped into an algebraic form in which some algebraic operators have predicates, headers, etc, that contain other algebraic operators. More specifically, for each record, δ, of the stream passing through the outer operator of a nested query, the inner query is evaluated by concatenating δ with each record of the inner query stream.

An unnest path is, and operator predicates may contain, a path expression $v/path$, $v/path/text()$, or $v/path/data()$, where v is a record attribute in the operator's input sequence and *path* is a simple XPath of the form:

$$\begin{aligned} path ::= &\ A \\ \mid &\ A/path \end{aligned}$$

for a tag name A. The unnest operation is the only mechanism provided for traversing an XML tree structure. It uses function $\mathcal{P}$, which is defined over paths as follows:

$$\begin{aligned}
\mathcal{P}(t, v/path) &= \mathcal{P}'(t.v, path) \\
\mathcal{P}'(\text{<A>}x \text{ </A>}, A/path) &= \mathcal{P}'(x, path) \\
\mathcal{P}'(\text{<A>}x \text{ </A>}, A) &= \{\ \text{<A>}x \text{ </A>}\ \} \\
\mathcal{P}'(x_1\, x_2, path) &= \mathcal{P}'(x_1, path) \cup \mathcal{P}'(x_2, path) \\
\mathcal{P}'(t, path) &= \emptyset \qquad\qquad\qquad \text{otherwise}
\end{aligned}$$

The extraction operator, ρ, gets an XML document, T, and returns a singleton list whose unique element contains the entire XML tree. Selection (σ), projection (π), merging ($\cup$), and join ($\bowtie$) are similar to their relational algebra counterparts, while unnest (μ) and nest ($\varGamma$) are based on the nested relational algebra. The reduce operator, $\varDelta$, is used in producing the final result of a query/subquery, such as in aggregations and existential/universal quantifications. For example, the XML universal quantification **every** $v **in** $x/A **satisfies** $v/A/data()>5 can be captured by the $\varDelta$ operator, with $\oplus = \wedge$ and *head*$= v/A/data()>5$. Collection query results can be constructed by $\varDelta$ by using a collection monoid, such as $\cup$. Like the XQuery predicates, the predicates used in our XML algebraic operators have implicit existential semantics related to the (potentially multiple) values returned by path expressions. For example, the predicate $v/A/data()>5$ used in the previous example has the implicit semantics $\exists x \in v/A/data() : x>5$, since the path $v/A/data()$ may return more than one value. Finally, even though selections and projections can be expressed in terms of $\varDelta$, for convenience, they are treated as separate operations. Please refer to [11], for a complete treatment of translating queries written in XQuery to our algebra, and for some query normalization and optimization rules.

3.1 Example

To illustrate the usage of the operators in our algebra, we consider the following XQuery, which returns the list of vendors selling HP PDAs:

```
   for $b in document(commodities)//vendor//item
 where $b/name = "PDA"
   and $b/make = "HP"
return <vendor> { $b/../../name } </vendor>
```

Its corresponding equivalent algebraic form is:

$$\Delta^{\cup,h}(\sigma_{p_2}(\sigma_{p_1}(\mu^{b,v/\text{items/item}}(\mu^{v,c/\text{commodities/vendor}}(\rho_c(T))))))$$

where

$$\begin{aligned} T &= \text{document(commodities)} \\ h &= \text{element}(\text{“vendor”}, v/\text{name}) \\ p_1 &= b/\text{name} = \text{“PDA”} \\ p_2 &= b/\text{make} = \text{“HP”} \end{aligned}$$

The extraction operator, ρ_c, extracts the complete XML tree from the commodities document and binds it to the variable c, while the unnest operator, $\mu^{v,c/\text{commodities/vendor}}$, traverses the document using the path c/commodities/ vendor, and binds the elements obtained by the unnesting operation to the variable v. These elements are subsequently unnested by $\mu^{b,v/\text{items/item}}$, binding the variable b to each item. The selection operators, σ_{p_2} and σ_{p_1}, filter the elements based on the predicates $b/\text{name} = \text{“PDA”}$ and $b/\text{make} = \text{“HP”}$ respectively. The element construction function, element(“vendor”, v/name), constructs an XML element with tag name “vendor” and content v/name, while the reduce operator $\Delta^{\cup,h}$ unions together the elements returned by the element function.

4 An Algebra for Fragmented XML Stream Data

The algebraic operators for streamed XML data take stream fragments as input and produce stream fragments as output. The fragments are streamed through the various operators in a pipelined fashion. While all the incoming fragments are examined, some of them are discarded when the operator predicate evaluates to false. Some fragments may not contain enough information to be evaluated by a particular operator, due to the presence of holes. When an operator has insufficient information to validate a fragment, it suspends the processing of this fragment until the relevant fillers arrive.

Before we present the details of the algebraic operators, we give some useful definitions.

Since a streamed query may operate on multiple input streams, the client must maintain one tag structure for each input stream. For an input stream, s, function TS(s) returns the tag structure of s (an XML tree). Furthermore, for each input stream s and for each tsid m in $TS(s)$, function $\mathrm{TSID}(s,m)$ returns the subelement of this tag structure with id $= m$. Expressed in XPath, it is equal to:

$$\mathrm{TSID}(s,m) = \mathrm{TS}(s)//\mathrm{tag}[@\mathrm{id} = \text{“}m\text{”}]$$

For example, $\mathrm{TSID}(s,7)$ evaluates to the tag structure element <tag name= “make” id=“7”/>, for the tag structure corresponding to the commodities stream s presented earlier. Given a tag structure s and a path expression $path$, $\mathcal{Q}(s, path)$ returns the set of tag structures reachable by $path$:

$$\begin{aligned}
&\mathcal{Q}(\text{<tag name=“A” ...>}y\text{</tag>}, A/path)\\
&\qquad\qquad = \mathcal{Q}(y, path)\\
&\mathcal{Q}(\text{<tag name=“A” ...>}y\text{</tag>}, A)\\
&\qquad = \{\,\text{<tag name=“A” ...>}y\text{</tag>}\,\}\\
&\mathcal{Q}(x_1\, x_2, path) = \mathcal{Q}(x_1, path) \cup \mathcal{Q}(x_2, path)\\
&\quad\mathcal{Q}(x, path) = \emptyset \qquad\qquad\qquad \text{otherwise}
\end{aligned}$$

The intermediate results between our stream algebraic operators are records of the form: $<v_1 = e_1, \ldots, v_n = e_n>$, where v_i are range variables introduced by the algebraic operators and e_i are XML fragments. The hole ID m in $F(m, n, x)$ can be -1 to indicate that this filler is generated by one of the operators and does not fill any hole. Each range variable v is associated with a set of tag structures, called $\mathrm{DOMAIN}(v)$, which can be statically determined (at compile-time) from the algebraic operator tree and the tag structures of the input streams using the function $\mathcal{Q}$: Each range variable v ranges over the result of a path expression $w/path$, thus,

$$\mathrm{DOMAIN}(v) = \{\, y \,|\, x \in \mathrm{DOMAIN}(w),\ y \in \mathcal{Q}(x, path)\,\}$$

Content Restriction: For each intermediate result record of the form $< \ldots, v_i = F(m,n,x), \ldots >$, the condition, $\exists x \in \mathrm{DOMAIN}(v_i) :\ x//\mathrm{tag}/@\mathrm{id} = \text{“}n\text{”}$, must hold. That is, the tag structure of the fragment streamed through the input and associated with the range variable v_i is not necessarily equal to one of the tag structures of v_i, but, more generally, it can be a descendant of one of them. If a record does not satisfy this restriction, it is suspended (explained below).

Each n-ary algebraic operator *op* in the algebraic tree of a query is associated with $n+1$ binding lists ζ_i, $0 \le i \le n$, one for for each input and one for the output, ζ_0, which maps fragment IDs to fragment contents. If a fragment from the operator's i'th input stream cannot be processed due to fragment holes, then the fragment is suspended in ζ_i. When a filler comes that fills one of the holes of a suspended fragment, then the fragment is awaken again and re-examined.

Function $\mathcal{P}'$ (defined in Section 3) is extended to handle holes:

$$\begin{aligned}
\mathcal{P}'(\text{<hole id=“m” .../>}, path) &= \mathcal{P}'(\zeta_i(m), path)\\
&\qquad\qquad\qquad \text{if}\, m \in \zeta_i\\
\mathcal{P}'(\text{<hole .../>}, path) &= \{\ \perp\ \}\ \ \text{otherwise}
\end{aligned}$$

That is, $\mathcal{P}'(x, path)$ returns a set of elements, where each element can be either a $\perp$ or an XML element. If there is at least one $\perp$ in $\mathcal{P}'(x, path)$, the *path* cannot be completely evaluated against x due to holes in x.

Given an intermediate result record t and a predicate *pred* over this record, $\mathcal{P}(t, pred)$ can return a true, false, or $\perp$ (undefined) value. That is, $\mathcal{P}'(x, path)$ returns a set of elements, where each element can be either a $\perp$ or an XML element. A predicate is in general a boolean formula that combines simple predicates using boolean operators ($\wedge$, $\vee$, etc). A simple predicate typically compares two XPaths or one XPath with a value. If *pred* is a simple predicate and there is a path $v/path$ in *pred* such that $\perp \in \mathcal{P}(t, v/path)$, then $\mathcal{P}(t, pred)$ evaluates to $\perp$; otherwise, $\mathcal{P}(t, pred)$ is true if for each path $v/path$ in *pred*, there is an XML value $x \in \mathcal{P}(t, v/path)$ such that *pred* evaluates to true when $v/path$ is replaced by x; otherwise, it is false. For the boolean operators, we use three-value logic. For example, false $\vee$ false = false, true $\vee\, x = x \,\vee$ true = true, otherwise $x \vee y = \perp$.

The extension of the path evaluation function $\mathcal{P}$ to handle fillers and holes forms the basis for the extension of our algebraic operators to operate on the fragmented XML model. We will now see how each operator is extended to handle the holes and fillers in XML streams. Note that each operator is producing output as a sequence of records as illustrated by the following BNF,

$$s := <v_1 = e_1, \ldots, v_n = e_n> \; ; \; s \mid eos$$

Where $<v_1 = e_1, \ldots, v_n = e_n>$ is a record output from the operators, as discussed earlier in this section. This accounts for the pipelined operation of the algebraic operators while interrogating the stream, modeled as a sequence of fragments.

We will now show how some of the operators for stored XML algebra have been extended for streamed XML.

4.1 The Extraction Operator ρ_v

The extraction operator extracts the fragments from the stream and identifies each of these fragments with the range variable v. Every input fragment satisfies the content restriction since the tag structure of v is the root tag structure. Since this operator does not use paths, no fragments need to be suspended. The semantics of this operator is straightforward:

$$\begin{array}{c} [\![\rho_v(t\,;\gamma)]\!]_\delta \rightarrow < v = t > ; \, [\![\rho_v(\gamma)]\!]_\delta \\ [\![\rho_v(eos)]\!]_\delta \rightarrow eos \end{array}$$

where the environment δ is the current record under consideration, as in the stored XML algebra.

4.2 The Selection Operator σ_{pred}

The selection operator filters the fragments from the stream based on the predicate specified. Since selection is a unary operator, it uses one environment ζ_1 for the input stream, which is always empty, and one environment ζ_0 for the output stream. If the selection predicate *pred* cannot be evaluated completely against the current tuple t of the input stream, then the tuple is suspended in ζ_0:

$$\zeta_0 \mathrel{+}= \{t\} \text{ if } \mathcal{P}(\delta \circ t, \mathit{pred}) = \bot$$

where operation $\zeta_0 \mathrel{+}= ts$ adds a set of tuples ts to ζ_0. Thus, selection lets the tuple t pass to the output stream only if *pred* evaluates to true:

$$\begin{aligned}
[\![\sigma_{pred}(t\,;\,\gamma)]\!]_\delta &= \text{if } \mathcal{P}(\delta \circ t, \mathit{pred}) = \text{true} \\
&\qquad \text{then } t\,;\, [\![\sigma_{pred}(\gamma)]\!]_\delta \\
&\qquad \text{else } [\![\sigma_{pred}(\gamma)]\!]_\delta \\
[\![\sigma_{pred}(eos)]\!]_\delta &= eos
\end{aligned}$$

If $\mathcal{P}(\delta \circ t, \mathit{pred})$ evaluates to *true*, then the fragment record t is output. The operator then recursively processes the remaining stream of records. If $\mathcal{P}(\delta \circ t, \mathit{pred})$ evaluates to false (failed predicate) or $\bot$ (presence of a hole), then the fragment record t is not released to the output stream. Additionally, at the arrival of *eos*, the buffer ζ_0 is cleared, i.e $\zeta_0 = \emptyset$.

4.3 The Projection Operator $\pi_{v_1,\ldots,v_n}$

The projection operator uses the projection variables to select attributes from the current tuple of the input stream:

$$\begin{aligned}
&[\![\pi_{v_1,\ldots,v_n}(t\,;\,\gamma)]\!]_\delta \\
&\qquad = < v_1 = t.v_1, \ldots, v_n = t.v_n >\,;\, [\![\pi_{v_1,\ldots,v_n}(\gamma)]\!]_\delta \\
&[\![\pi_{v_1,\ldots,v_n}(eos)]\!]_\delta = eos
\end{aligned}$$

4.4 The Merge Operator $\bigcup$

The merge operator propagates the input tuples from either one of the two input streams into the output stream (without removing duplicates):

$$[\![(t_1\,;\,\gamma_1) \bigcup \gamma_2]\!]_\delta = t_1\,;\, [\![\gamma_1 \bigcup \gamma_2]\!]_\delta$$

$$[\![\gamma_1 \bigcup (t_2\,;\,\gamma_2)]\!]_\delta = t_2\,;\, [\![\gamma_1 \bigcup \gamma_2]\!]_\delta$$

$$[\![eos \bigcup \gamma]\!]_\delta = [\![\gamma \bigcup eos]\!]_\delta = \gamma$$

Note that the semantic interpretation of the merge operation captures effectively the processing of streamed fragments when data is available in either of the streams, or possibly on both at the same time.

4.5 The Join Operator $\bowtie_{pred}$

The streamed join operator performs a join between the fragments of two streams γ_1 and γ_2 based on the predicate *pred*. It suspends all the tuples from both input streams because each fragment from either stream needs to be preserved since it may be joined with incoming fragments from the other stream. If the evaluation of the join predicate *pred* over a pair of records t_1 and t_2 is undetermined (i.e., $\perp$) due to the presence of holes, the concatenation $t_1 \circ t_2$ is suspended in the binding list of the output stream, ζ_0. Furthermore, tuples from the left stream are suspended in ζ_1:

$$\begin{aligned} &[\![(t_1\,;\,\gamma_1) \bowtie_{\mathit{pred}} \gamma_2]\!]_\delta \\ &\qquad = \{\, t_1 \circ t_2 \mid t_2 \in \zeta_2,\ \mathcal{P}(\delta \circ t_1 \circ t_2, \mathit{pred}) = \text{true} \,\} \\ &\qquad\qquad\qquad ;\ [\![\gamma_1 \bowtie_{\mathit{pred}} \gamma_2]\!]_\delta \end{aligned}$$

$$\begin{aligned} \zeta_1 &\mathrel{+}= t_1 \\ \zeta_0 &\mathrel{+}= \{\, t_1 \circ t_2 \mid t_2 \in \zeta_2,\ \mathcal{P}(\delta \circ t_1 \circ t_2, \mathit{pred}) = \perp \,\} \end{aligned}$$

The tuples from the right stream are suspended in ζ_2:

$$\begin{aligned} &[\![\gamma_1 \bowtie_{\mathit{pred}} (t_2\,;\,\gamma_2)]\!]_\delta \\ &\qquad = \{\, t_1 \circ t_2 \mid t_1 \in \zeta_1,\ \mathcal{P}(\delta \circ t_1 \circ t_2, \mathit{pred}) = \text{true} \,\} \\ &\qquad\qquad\qquad ;\ [\![\gamma_1 \bowtie_{\mathit{pred}} \gamma_2]\!]_\delta \end{aligned}$$

$$\begin{aligned} \zeta_2 &\mathrel{+}= t_2 \\ \zeta_0 &\mathrel{+}= \{\, t_1 \circ t_2 \mid t_1 \in \zeta_1,\ \mathcal{P}(\delta \circ t_1 \circ t_2, \mathit{pred}) = \perp \,\} \end{aligned}$$

When an *eos* is received in stream γ_1, the contents of buffer ζ_2 can be flushed, since they are not needed anymore (there cannot be any more fragments of XML from stream ζ_1 that would need to be joined with fragments in ζ_2). The converse is true when an *eos* is received in stream γ_2, wherein buffer ζ_1 can be flushed. After both streams receive an *eos*, an *eos* is embedded to the output stream. Finally, and more importantly, the binding lists ζ_0 is reduced when a filler of a hole in a fragment in ζ_0 arrives from the input streams, which causes the tuple that contains this fragment to be reconsidered by the join operator.

The semantics of our join operator resembles the symmetric join algorithm [23]. To evaluate the equijoin $X \bowtie_{x.A=y.B} Y$ using a symmetric join, we have to maintain two hash tables in memory that contain the same number of buckets, one for stream X with a hash function based on $x.A$ and one for the stream Y based on $y.B$. When a tuple, x, arrives in the stream X, it is inserted in the X hash table and is joined with all the tuples in the Y hash table that have the same hash key. The matched pairs are immediately streamed into the output. A similar operation is performed when a new tuple arrives in the stream Y.

4.6 The Unnest Operator $\mu^{v,path}_{pred}$

The unnest operator provides the means to traverse the tree based on the path expression *path* and the predicate *pred*. It unnests the XML fragments based on the path expression and then applies the predicate validation:

$$\begin{aligned}
&\llbracket \mu^{v,path}_{pred}(t\,;\,\gamma)\rrbracket_\delta \\
&\quad= \{\, t\circ < v = w > \mid w \in \mathcal{P}(\delta\circ t, path),\ w \neq \bot, \\
&\qquad\qquad\qquad \mathcal{P}(\delta\circ t\circ < v = w >, pred) = \text{true}\,\} \\
&\qquad ;\ \llbracket \mu^{v,path}_{pred}(\gamma)\rrbracket_\delta
\end{aligned}$$

$$\begin{aligned}
\zeta_0 \mathrel{+}= {} & \{\, t\circ < v = w > \mid w \in \mathcal{P}(\delta\circ t, path), \\
& \qquad\quad \mathcal{P}(\delta\circ t\circ < v = w >, pred) = \bot\,\}
\end{aligned}$$

$$\llbracket \mu^{v,path}_{pred}(eos)\rrbracket_\delta = eos$$

If the unnesting path or the predicate evaluates to $\bot$, then the unnesting pair (the current record concatenated with one of the results of applying *path* over the record) is suspended in ζ_0. If an *eos* is received, we return it and we flush the output buffer ζ_0.

4.7 The Reduce Operator $\boldsymbol{\varDelta^{\oplus,head}_{pred}}$

The reduce operator applies the header to each tuple and merges the results using $\oplus$:

$$\begin{aligned}
&\llbracket \varDelta^{\oplus,head}_{pred}(t\,;\,\gamma)\rrbracket_\delta \\
&\quad= \text{if}\ \mathcal{P}(\delta\circ t, pred) = \text{true}\ \wedge\ \mathcal{P}(\delta\circ t, head) \neq \bot \\
&\qquad\quad \text{then}\ \mathcal{P}(\delta\circ t, head)\ \oplus\ \llbracket \varDelta^{\oplus,head}_{pred}(\gamma)\rrbracket_\delta \\
&\qquad\quad \text{else}\ \llbracket \varDelta^{\oplus,head}_{pred}(\gamma)\rrbracket_\delta
\end{aligned}$$

$$\llbracket \varDelta^{\oplus,head}_{pred}(t\,;\,\gamma)\rrbracket_\delta = eos$$

If either the predicate or the header is unable to evaluate, the current tuple is suspended:

$$\zeta_0 \mathrel{+}= \{t\}\ \text{if}\ \mathcal{P}(\delta\circ t, pred) = \bot \vee \mathcal{P}(\delta\circ t, head) = \bot$$

4.8 The Nest Operator $\boldsymbol{\varGamma^{v,\oplus,head}_{group,pred}}$

The nest operator itself does not produce any output, since it is blocked:

$$\llbracket \varGamma^{v,\oplus,head}_{group,pred}(t\,;\,\gamma)\rrbracket_\delta = \llbracket \varGamma^{v,\oplus,head}_{group,pred}(\gamma)\rrbracket_\delta$$

$$\llbracket \varGamma^{v,\oplus,head}_{group,pred}(eos)\rrbracket_\delta = eos$$

If any of the head, group, or pred cannot be evaluated completely, the tuple t is suspended in ζ_1. The ζ_0 binding list contains one tuple for each group. If a new group is found, then a new tuple is inserted in ζ_0; otherwise, the head of the current tuple is accumulated in the group tuple:

```
if $\mathcal{P}(\delta \circ t, pred)$ = true:
  find $x \in \zeta_0 : \mathcal{P}(\delta \circ x, group) = \mathcal{P}(\delta \circ t, group)$;
  if none exists:
    then $\zeta_0 \mathrel{+}= \mathcal{P}(\delta \circ t, group) \circ < v = \mathcal{P}(\delta \circ t, head) >$
    else $x.v = x.v \oplus \mathcal{P}(\delta \circ t, head)$.
```

The groups can only be flushed after the end of the input stream.

4.9 An XQuery Example Using the Streamed XML Algebra

To illustrate the usage of the operators in our streamed XML algebra, we consider the XQuery presented in Section 3.1, which returns the list of names of vendors selling HP PDAs evaluated against the fragments from the commodities stream, as shown in Section 2. Let f_1, f_2, f_3 correspond to the fragments 1, 2 and 3. The commodities stream can be visualized as a sequence of the fragments $f_1; f_2; f_3; ...; eos$. When the fragments are streamed through the operators, the unnest operator, $\mu^{b,v/\text{items/item}}$, suspends the fragment f_1, since it encounters a hole with $id = 10$ during its path traversal. When the fragment f_2 arrives, with $id = 10$, the suspended fragment is re-triggered, and f_2 is streamed through. This fragment is then passed through the second unnest operator as well as the selection operators, σ_{p_2} and σ_{p_1}, and is added to the result stream. Note that fragment f_1 is still suspended due to the presence of another hole with $id = 20$. When the fragment f_3 arrives, with $id = 20$, it is similarly streamed through the unnest operator, however, is filtered out in the selection stage. The *eos* stream element flushes the buffers. Note that the structural context of a fragment can be deduced with the help of the *tsid* attribute in the fragments and the tag structures transmitted by the server.

4.10 Equivalence between the Stored and the Streamed XML Algebra

Given an XML query, its algebraic tree based on our streamed XML algebra is exactly the same as that for our stored XML algebra, since both algebras use identical algebraic operators. Therefore, to prove the equivalence theorem in Section 2.3, we need simply to prove that for each n-ary algebraic operator *op* of the stored algebra and its equivalent *op'* of the streamed algebra, we have:

$$op(\mathcal{T}(\gamma_1), \ldots, \mathcal{T}(\gamma_n)) \;=\; \mathcal{T}(op'(\gamma_1, \ldots, \gamma_n))$$

for arbitrary finite streams γ_i. That way, by starting from the algebraic tree in the stored algebra of an n-ary query q, we can build the algebraic tree in the streamed algebra using the above natural transformations, since the $\mathcal{T}$ translations can propagate bottom-up in the query tree, yielding at the end the algebraic tree for the streamed query q' that satisfies:

$$q(\mathcal{T}(\gamma_1), \ldots, \mathcal{T}(\gamma_n)) \;=\; \mathcal{T}(q'(\gamma_1, \ldots, \gamma_n))$$

With this equivalence of the algebraic operators for the fragmented XML stream as for stored XML data, it is beneficial to operate on fragments as and when they arrive, instead of waiting to materialize the complete document for two main reasons, firstly, processing can be continuous, pipelined and timely, secondly, the fragments that do not satisfy the predicates can be safely discarded as soon as possible thereby conserving memory.

5 Related Work

There are many recent projects related to query processing on data streams, which are overviewed elsewhere [2, 3].

The hole-filler model for XML data has been proposed in [19] in the context of pull-based content navigation over mediated views of XML data from disparate data sources. In our framework, the hole-filler model is used in a push-based model, for fragmenting XML data to be sent to clients for selective query processing. Our main motivation is to relieve the load on the server and leverage the processing power in client devices. Based on previous experience with traditional databases, queries can be optimized more effectively if they are first translated into a suitable internal form with clear semantics, such as an algebra or calculus. There are already many proposals for algebras on semi-structured and XML data, including an algebra based on structural recursion [5], YATL [9, 8], SAL [4], x-algebra [12], TAX [17], and the Niagara algebra [22]. In contrast to these algebras, with the possible exception of the Niagara algebra, our work is based on the nested relational algebra, since our focus is on pipelining the algebraic operators using main-memory, relational evaluation algorithms. In addition, there is a recent proposal for a data model and an XQuery algebra by the W3C committee [10], whose main purpose is in expressing well-formed semantics, such as type inference. This algebra is basically a core subset of XQuery and, unlike conventional database algebras, does not address performance issues.

Our goals are similar to those of the Niagara Continuous Query Processing project, NiagaraCQ [7]. A recent work by this group proposed the use of punctuations for handling blocking operators [20]. A punctuation is a hint transmitted by a server to clients along with the data to indicate properties about the data. One example of a punctuation is the indication that all prices of stocks starting with 'A' have already been transmitted. Punctuations are properties that hold from the point of their transmission up-to the end of the stream, allowing us to view an infinite stream as a mixture of finite streams.

There is some work recently on using stream transducers to process continuous XML streams [18]. A transducer generalizes deterministic finite automata (DFAs) in that it allows the generation of output during a state transition. Even though DFAs have been shown to achieve a high throughput for XPath expressions and for non-blocking XQueries on single XML streams [14], it still to be shown that are also effective for multiple input streams, which require advanced main-memory join techniques that have already been successfully addressed by main-memory databases.

6 Conclusion

In this paper, we have illustrated the main theme of our work: evaluating the user defined queries on fragmented XML stream data in a continuous fashion, as opposed to materializing the stream and then evaluating the queries on the complete XML. We started by motivating the need for fragmented XML dissemination and subsequent query evaluation on streamed fragments. Our query algebra for processing of fragmented XML data is modeled in the lines of that for complete XML data, hence the query optimizations proposed in our previous work can be applied to the fragmented XML data processing as well. The streamed fragments are processed by our algebraic operators and are then combined to form the final result. Since the data is continuous and so are the queries, we continuously update the result to reflect the data and the query predicates.

References

1. M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB 2000*.
2. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *PODS 2002*, pages 1–16.
3. S. Babu and J. Widom. Continuous Queries Over Data Streams. *SIGMOD Record*, 30(3):109–120, September 2001.
4. C. Beeri and Y. Tzaban. SAL: An Algebra for Semistructured Data and XML. In *WebDB 1999*, pages 37–42.
5. P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *SIGMOD 1996*, pages 505–516.
6. D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: A Query Language for XML. W3C Working Draft. Available at `http://www.w3.org/TR/xquery/`, 2000.
7. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD 2000*, pages 379–390.
8. V. Christophides, S. Cluet, and J. Siméon. On Wrapping Query Languages and Efficient XML Integration. In *SIGMOD 2000*, pages 141–152.
9. S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your Mediators Need Data Conversion! In *SIGMOD 1998*, pages 177–188.
10. D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft. Available at `http://www.w3.org/TR/query-algebra/`, 2002.
11. L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query Processing of Streamed XML Data. In *CIKM 2002*, pages 126–133.
12. M. Fernandez, J. Simeon, and P. Wadler. An Algebra for XML Query. In *FST TCS 2000*.
13. D. Florescu, A. Levy, and A. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, 1998.
14. T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *ICDE 2003*.

15. A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *SIGMOD 2003*, pages 419–430.
16. Z. Ives, A. Levy, and D. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical Report, University of Washington, 2000, UW-CSE-2000-05-02.
17. H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In *DBPL 2001*, pages 149–164.
18. B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *VLDB 2002*.
19. B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. Navigation-driven Evaluation of Virtual Mediated Views. In *EDBT 2000*, LNCS 1777.
20. P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Online analysis and Querying of Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, May-June 2003.
21. T. Urhan and M. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(3):27–33, 2000.
22. S. Viglas, L. Galanis, D. DeWitt, D. Maier, and J. Naughton. Putting XML Query Algebras into Context. Unpublished manuscript, 2002.
23. A. Wilschut and P. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *PDIS 1991*, pages 68–77.

Updates and Incremental Validation of XML Documents

Béatrice Bouchou and Mírian Halfeld Ferrari Alves

Université de Tours - Laboratoire d'Informatique
Antenne Universitaire de Blois
3 place Jean Jaurès
41000, Blois, France
{bouchou,mirian}@univ-tours.fr

Abstract. We consider the incremental validation of updates on XML documents. When a *valid* XML document (*i.e.*, one satisfying some constraints) is updated, it has to be verified that the new document still conforms to the imposed constraints. Incremental validation of updates leads to significant savings on computing time when compared to brute-force validation of an updated document from scratch.

This paper introduces a correct and complete set of update operations that can be integrated in an XML manipulation language. Indeed, any document generated by a composition of our update operations is valid, and, every valid document can be generated by a composition of our update operations (from the empty document). To accept an update, the validity of the result is checked first (without any change on the original document). Validation tests are performed incrementally, *i.e.*, only the validity of the part of the document directly affected by the update is checked. Changes to the original document are effectively performed *only* when the update is accepted.

1 Introduction

We present a method for incrementally validating updates on an XML document. We assume a data-exchange environment where an XML document should respect schema constraints. When a valid XML document is updated, it has to be verified that the new document still conforms to the imposed schema. Validation from scratch requires reading the entire document after each update. An incremental method is undoubtedly very useful, in particular when we consider that the evolution of XML as an exchange format depends on its capability to support not only queries but also updates.

We view an XML document as a structure $\mathcal{T}$ composed of an unranked labeled tree t (*i.e.*, a tree whose nodes have no fixed -or ranked- arity) and functions *type* and *value*. The function *type* indicates the type of a node (*element*, *attribute* or *data*). The function *value* gives the value associated with a leaf (a data node). Figure 1 shows part of the labeled tree representing the document used in our examples. Each node has a position and a label (for instance, position 0 is associated with label $Cust$). From this figure we see that an XML element has both its sub-elements and attributes as children in the tree. Elements and attributes associated with arbitrary text have a child labeled *data*. Attribute labels are depicted with a preceding @.

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 216–232, 2004.

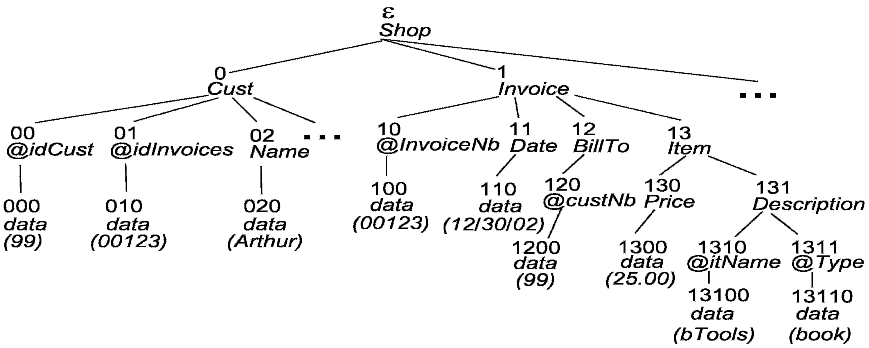


Fig. 1. Labeled tree t representing an XML document.

XML documents should respect a schema $\mathcal{D}$ which corresponds to attribute and element restrictions. We see the schema $\mathcal{D}$ as a structure composed of a bottom-up tree automaton and some "extra" information about attribute values, both introduced in [5] where only validation from scratch is considered. Validating schema constraints means to execute the tree automaton over the labeled tree t. This computation results in another labeled tree r (called a running tree) with the same positions as t, but labeled with the *states* of the tree automaton, as illustrated in Example 1 below. Roughly, a state q is assigned to a position p in r if the children of p in t verify the element and attribute constraints established by the tree automaton.

Example 1. Figure 2 shows a running tree resulting from the execution of a given tree automaton over the tree t of Figure 1. To illustrate the execution of this tree automaton, suppose that it has the transition rule

$$Invoice, \{\{q_{invoiceNb}\}, \emptyset\}, q_{Date}\ q_{BillTo}\ q^{*}_{Item} \rightarrow q_{Invoice}.$$

This rule states that a position p, labeled $Invoice$ in t, can be associated with the state $q_{Invoice}$ in r if the following attribute and element constraints are respected:
(i) position p has a required attribute child $invoiceNb$ (*i.e.*, the first child of p in r is associated with $q_{invoiceNb}$) and
(ii) children of p that are elements respect the regular expression $Date\ BillTo\ Item^{*}$ (*i.e.*, the second and the third children of p in r are associated with q_{Date} and q_{BillTo}, respectively; the other right children (if they exist) are associated with q_{Item}).
For instance, these constraints are respected by position 1 in r and thus 1 is associated with $q_{invoice}$ (Figure 2). The automaton executes bottom-up, considering each position and the transition rule that applies to it. A tree automaton accepts the document tree if and only if the root of the corresponding running tree is labeled with a *final* state. □

Given a valid XML document, an update is done taking into account the following features:

- Updates are seen as changes to be performed on the tree representation $\mathcal{T}$ of an XML document. We do not consider here the translation between an XML document and its tree representation, this is done by well known tools such as SAX and DOM.

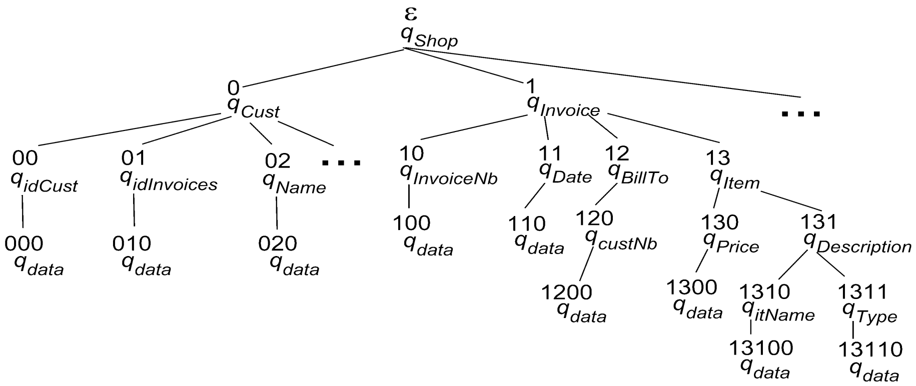


Fig. 2. Running tree r resulting from the execution of a tree automata over t.

• Only updates that preserve the validity of the document are accepted. If the update violates a constraint, then it is rejected and the XML document remains unchanged.
• The acceptance of an update relies on *incremental validation* tests, *i.e.*, only the validity of the part of the original document directly affected by the update is checked.

Based on the above points, we introduce a set of update operations capable of inserting, deleting or modifying parts of tree representing an XML document. Before accepting an update, we perform *incremental* validity tests which consist of verifying the validity of a small part of the document tree. If the desired update concerns position p, we just check if the subtree rooted at p's father continues to respect the validity conditions. Let $\mathcal{A}$ be the automaton in $\mathcal{D}$. Let δ be a transition rule of $\mathcal{A}$ that associates a position u, labeled a, with state q_a. An update at position p (a child of u) changes the sequence of states associated with u's children by the automaton. Considering the new children of u, we need to verify if they still respect the constraints established by δ. If these constraints are respected, δ can be applied and the label q_a is associated with u, otherwise the intended update is rejected since it violates constraints. To efficiently verify if δ applies to the new children of u, we only build a temporary *sequence of new states* of u's children. The following example illustrates this process.

Example 2. Let $\mathcal{D}$ be a schema containing an automaton $\mathcal{A}$ with the following transition rules (among others):

(1)Item, $\{\emptyset, \emptyset\}$, $q_{Price}\ q_{Description} \rightarrow q_{Item}$
(2)Price, $\{\emptyset, \emptyset\}$, $q_{data} \rightarrow q_{Price}$
(3)data, $\{\emptyset, \emptyset\}$, $\emptyset \rightarrow q_{data}$

Now, we assume the valid document of Figure 1 describing the customers and invoices of a shop. Each invoice contains the price and the description of the items bought by a customer. We consider the item depicted at position 13 and we assume the insertion of another price for this item. This operation corresponds to the insertion of a labeled tree t_1 (having positions ϵ and 0 associated with labels $Price$ and $data$, respectively) at position 131 of the tree t (Figure 1). The labeled tree t' in Figure 3 represents the requested change over t.

The verification of the update consists in: (i) considering that the update is performed (without performing it yet) and (ii) verifying if the state q_{Item} can still be associated with position 13 (131's father) by analyzing the unique transition rule whose head is q_{Item}.

To this end, we are going to build the sequence of states associated with 13's children. To better illustrate our example, we consider the subtree of t (Figure 1) whose root is at position 13 and its requested updated version (the corresponding subtree on Figure 3). We assume the bottom-up execution of $\mathcal{A}$ over t_1. We apply rule 3 over the leave of t_1 to obtain the state q_{data}. This leave shall be at position 1310 if the update is accepted (see requested updated tree partly depicted by Figure 3). Then, we apply rule 2 over the root of t_1 to obtain q_{Price}. The root of t_1 is at position 131 of the requested updated tree (Figure 3). Note that this update does not concern the subtrees on the left of position 131: nothing changes for the subtree rooted at position 130. Moreover, the subtrees on the right have just been shifted (see the subtree now rooted at position 132 of Figure 3). In other words, the update does not affect position 130 (associated with q_{Price}) and position 132 is the result of a shift (a new position, but associated with an "old" state, *i.e.*, one computed before the update). Thus, we only have to calculate the state associated with position 131 in order to obtain the complete new sequence of children states for position 13.

Now, we consider rule (1). It can be applied to position 13 if all the following conditions hold: (i) $t(13) = Item$, (ii) for all children pos of 13 we have $type(t, pos) \neq attribute$ and (iii) the concatenation of the labels associated with 13's children composes a word that corresponds to the regular expression $q_{Price}\ q_{Description}$. In our case this concatenation is $q_{Price}\ q_{Price}\ q_{Description}$ (positions 130, 131 and 132, respectively). This word does not match the regular expression $q_{Price}\ q_{Description}$, so condition (iii) does not hold. Rule 1 cannot be applied to position 13. The update is rejected, since it violates validity.

Notice that accepting or rejecting an update depends on the schema being considered. For instance, if we consider a schema $\mathcal{D}'$ similar to $\mathcal{D}$ except for transition rule (1) that is replaced by: Item, $\{\emptyset, \emptyset\}$, $q^*_{Price}\ q_{Description} \rightarrow q_{Item}$
then the insertion of t_1 at position 131 in t is accepted. Indeed, the concatenation $q_{Price}\ q_{Price}\ q_{Description}$ matches the regular expression $q^*_{Price}\ q_{Description}$. □

The main contributions of the paper are:

• The definition of a structure, called XML *dossier*, that formalizes and summarizes all the features necessary to the update validation.

• A correct and complete set of update operations. Indeed any XML dossier generated by a composition of our update operations is valid, and given a schema $\mathcal{D}$, every XML dossier valid with respect to $\mathcal{D}$ can be generated by a composition of these operations. The changes to an XML dossier due to an update are precisely defined. Four update operations are introduced, namely $insert$, $insertBefore$, $delete$ and $replace$.

• An incremental validation method that allows significant improvements over brute-force validation from scratch.

This paper is organized as follows: In Section 2, XML dossiers are defined and we discuss each component of this structure. In Section 3 we define the set of update op-

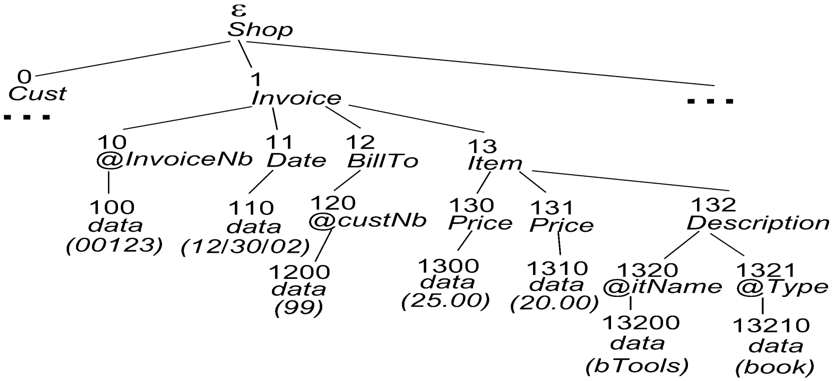


Fig. 3. Labeled tree t' representing the requested changes on t: an insertion at position 131.

erations and in Section 4 we show how incremental validation is performed. Finally, Section 5 concludes with related work and our perspectives for further research.

2 XML Dossiers

An XML dossier $\mathcal{X}$ contains all the components necessary to the validation of updates. It is a tuple $(\mathcal{D}, \mathcal{T}, \mathcal{R})$ where: $\mathcal{D}$ is a schema that defines attribute and element constraints, $\mathcal{T}$ is the tree representation of an XML document and $\mathcal{R}$ is a structure built from $\mathcal{D}$ and $\mathcal{T}$, over which (i) the validity of $\mathcal{T}$ with respect to $\mathcal{D}$ can be easily determined (only with few tests on values contained in $\mathcal{R}$), and (ii) the incremental validity test performed while updating is also easily applied.

2.1 XML Document Representation

There are different ways to view an XML document as a tree. Before introducing our choice of representation, we recall the notion of unranked Σ-valued trees [21]. Let $\mathbb{N}^*$ be the set of all finite strings of positive integers with the empty string ϵ as the identity. The following definition assumes that $dom(t) \subseteq \mathbb{N}^*$ is a nonempty set closed under prefixes[1], *i.e.*, if $u \preceq v$, $v \in dom(t)$ implies $u \in dom(t)$. Clearly, this set $dom(t)$ represents the set of nodes of t, uniquely identified with a Dewey like prefix schema.

Definition 1. – Σ-valued tree *t* [21]: Given an alphabet Σ, a nonempty Σ-valued tree t is a mapping $t : dom(t) \to \Sigma$ where $dom(t)$ satisfies: $j \geq 0, uj \in dom(t), 0 \leq i \leq j \Rightarrow ui \in dom(t)$. The set $dom(t)$ is also called the set of *positions* of t. We write $t(p) = a$, for $p \in dom(t)$, to indicate that the Σ-symbol associated with p is a. For each position p in $dom(t)$, $children(t, p)$ denotes the positions pi in $dom(t)$, and $father(t, p)$ denotes the *father* of p. Define an *empty tree* t as the one having $dom(t) = \emptyset$. □

[1] The *prefix relation* in $\mathbb{N}^*$, denoted by $\preceq$ is defined by: $u \preceq v$ iff $uw = v$ for some $w \in \mathbb{N}^*$.

Definition 2. – XML tree $\mathcal{T}$: Let $\Sigma = \Sigma_{ele} \cup \Sigma_{att} \cup \{data\}$ be an alphabet where Σ_{ele} is the set of element names and Σ_{att} is the set of attribute names. An XML tree is a tuple $\mathcal{T} = (t, type, value)$ where:

- t is a Σ-valued tree (*i.e.*, $t : dom(t) \rightarrow \Sigma$).
- $type$ and $value$ are functions defined as follows for a position $p \in dom(t)$:

$$type(t,p) = \begin{array}{lll} data & \text{if} & t(p) = data \\ element & \text{if} & t(p) \in \Sigma_{ele} \\ attribute & \text{if} & t(p) \in \Sigma_{att} \end{array}$$

$$value(t,p) = \begin{array}{l} setval \subset \mathbf{D} \text{ if } type(t,p) = data \\ undefined \text{ otherwise} \end{array}$$

where **D** is an infinite (recursively enumerable) domain. □

In Figure 1 we have, for instance, $type(t, 13) = element$ and $value(t, 1300) = \{25.00\}$.

To define update operations we need the notions of *frontier* and *insert frontier*. The frontier corresponds to the set of leaves while the insert frontier is the set of positions (not in $dom(t)$) where the simple insertion of new subtrees is possible.

Definition 3. – Frontier and insert frontier of a finite tree *t*: Given a tree t and considering $i \in \mathbb{N}$, the *frontier* of t, denoted by $fr(t)$ is defined by $fr(t) = \{u \in dom(t) \mid \neg\exists\ i \quad \text{such that } ui \in dom(t)\}$ while the *insert frontier* of t, denoted by $fr^{ins}(t)$ is defined by $fr^{ins}(t) = \{ui \notin dom(t) \mid u \in dom(t) \wedge [(i = 0) \vee ((i \neq 0) \wedge u(i-1) \in dom(t)]\}$. For an empty tree t, define $fr^{ins}(t) = \{\epsilon\}$. □

2.2 Schema Representation

We assume that XML views are built from different data sources according to a particular schema. In our approach, a schema $\mathcal{D}$ is specified by an extended non-deterministic bottom-up finite tree automaton (ENFTA) enhanced with an attribute table.

Definition 4. – Extended non-deterministic bottom-up finite tree automaton (ENFTA) [5]: An ENFTA over an alphabet Σ is a tuple $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ where Q is a set of states, $Q_f \subseteq Q$ is a set of final states and Δ is a set of transition rules of the form $a, S, E \rightarrow q$ where *(i)* $a \in \Sigma$; *(ii)* S is a set of two disjoint sets of states, *i.e.*, $S = \{S_{compulsory}, S_{optional}\}$ (with $S_{compulsory} \subseteq Q$ and $S_{optional} \subseteq Q$); *(iii)* E is a regular expression over Q and *(iv)* $q \in Q$. □

Definition 5. – Schema $\mathcal{D}$ for XML documents: Let $\Sigma = \Sigma_{ele} \cup \Sigma_{att} \cup \{data\}$ be a schema alphabet. A schema $\mathcal{D}$ for XML documents is a tuple $\mathcal{D} = (\mathcal{A}, Æ)$ where $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ is an ENFTA over Σ and $Æ[att\text{-}name, att\text{-}kind, ele]$ is an attribute table having one tuple for each pair $(att\text{-}name, ele)$ that associates an attribute $att\text{-}name \in \Sigma_{att}$ with an element $ele \in \Sigma_{ele}$. We assume that attribute kinds $att\text{-}kind$ in Æ are those possible in a DTD (*i.e.*, CDATA, ID, IDREF and IDREFS). □

Definition 4 extends classical tree automata in order to deal with trees with different kinds of nodes. In $\mathcal{T}$, the children of any position $p \in dom(t)$ can be classified into two groups: those that are unordered, corresponding to the attributes of the node, and those that are ordered, corresponding to the sub-elements.

In a schema $\mathcal{D}$, element constraints are expressed by regular expressions (part (iii) of $\mathcal{A}$'s transition rules). Attribute constraints imply two levels of specification. In the first level, for each element, the specification (part (ii) of $\mathcal{A}$'s transition rules) indicates the attributes that are obligatory ($S_{compulsory}$) and optional ($S_{optional}$). In the second level, for each attribute, the specification indicates its kind (attribute table Æ). Thus, as discussed in [5], the validity of attribute requires some tests on attribute values. These tests verify the uniqueness of identifier values (called ID values) in the whole document, and the existence of ID values corresponding to reference values (called IDREF or IDREFS values).

Example 3. We consider the schema $\mathcal{D} = (\mathcal{A}, Æ)$ (concerning customers and invoices in a shop) which has been used to build the running tree of Figure 2 from the tree of Figure 1. The schema alphabet Σ contains all the labels appearing in Figure 1.
The ENFTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ has $Q = \{q_a \mid a \in \Sigma\}$, $Q_f = \{q_{Shop}\}$ and twenty-one rules in Δ: four of them are presented in Examples 1 and 2.
The table Æ has the following tuples: { ⟨idCust, ID, Cust⟩, ⟨idInvoices, IDREFS, Cust⟩, ⟨invoiceNb, ID, Invoice⟩, ⟨custNb, IDREF, BillTo⟩, ⟨itName, CDATA, Description⟩, ⟨Type, CDATA, Description⟩ }. □

2.3 XML Documents Respecting a Schema

Given a schema $\mathcal{D} = (\mathcal{A}, Æ)$ and an XML tree $\mathcal{T} = (t, type, value)$, we want to verify if $\mathcal{T}$ respects the validity constraints imposed by $\mathcal{D}$. Consider first the execution of $\mathcal{A}$ over t. To assume a state q at position p, the automaton $\mathcal{A}$ performs the following tests:

1. If p has *attribute* children then their states should match those specified by the sets in S, namely $S_{compulsory}$ and $S_{optional}$, corresponding, respectively, to attributes that *must* appear in the tree and to those that *may* appear.
2. If p has *element* children then the concatenation of their states must belong to the language generated by the regular expression E.

We call *running tree* the Q-valued tree resulting from the execution of a tree automaton $\mathcal{A}$ over t.

Definition 6. – Running tree *r* [5]: Let t be a Σ-valued tree and $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ an ENFTA. A *running tree* r, corresponding to an execution of $\mathcal{A}$ over t, is a tree such that $dom(r) = dom(t)$ defined as follows: for each position p whose children are at positions $p0, \ldots, p(n-1)$ (with $n \geq 0$), $r(p) = q$ if all the following conditions hold:

1. $t(p) = a \in \Sigma$
2. There exists a transition $a, S, E \rightarrow q$ in Δ
3. There exists an integer $0 \leq i \leq (n-1)$ such that the children of p can be classified as follows:

(a) the positions $p0, \ldots, p(i-1)$ are members of a set *posAtt* (possibly empty) and
(b) the positions $pi, \ldots, p(n-1)$ are members of a set *posEle* (possibly empty) and
(c) every children of p is a member of *posAtt* or of *posEle* but no position is in both sets.

4. The tree r is already defined for p's children *i.e.*, $r(p0) = q_0, \ldots, r(p(n-1)) = q_{n-1}$.
5. The word $q_i \ldots q_{n-1}$, composed by the concatenation of the states associated with the positions in *posEle*, belongs to the language generated by E.
6. The sets of S respect the following properties:

(a) $S_{compulsory} \subseteq \{q_0, \ldots, q_{i-1}\}$ and
(b) $(\{q_0, \ldots, q_{i-1}\} \setminus S_{compulsory}) \subseteq S_{optional}$.

A running tree r is *successful* if $r(\epsilon)$ is a final state. □

From Definition 6, one can see that t is accepted by $\mathcal{A}$ if and only if r is *successful*. This is one of the conditions an XML tree $\mathcal{T}$ must respect to be valid relative to a schema $\mathcal{D}$.

Consider now the ID/IDREF constraints. During the run of $\mathcal{A}$ over t, bags of ID and IDREF(S) values are filled, according to the table Æ in $\mathcal{D}$. It is then straightforward to verify the ID/IDREF constraints. Precisely, we say that $\mathcal{T}$ respects $\mathcal{D}$ if all the following conditions hold:

C1- The running tree constructed according to Definition 6 from $\mathcal{D}$ and $\mathcal{T}$ is successful.
C2- The ID attributes in t are unique.
C3- The IDREF/IDREFS attributes refer to existing ID attributes.

To facilitate the verification of conditions C1-C3, we define the structure $\mathcal{R}$ containing the running tree r, together with ID and IDREF values.

Definition 7. – Run $\mathcal{R}$: Given a schema $\mathcal{D} = (\mathcal{A}, Æ)$ and $\mathcal{T} = (t, type, value)$, a run $\mathcal{R}$ is a tuple $\mathcal{R} = (r, V_{ID}, V_{IDREF})$ where r is the running tree (Definition 6) and V_{ID} and V_{IDREF}, called id-storage, are *bags* filled using Æ during the run of $\mathcal{A}$ on t, according to the steps below:

- If there exists a tuple in Æ that indicates that an attribute at position p has kind ID, then insert $value(t, p0)$ in V_{ID}.
- If there exists a tuple in Æ that indicates that an attribute at position p has kind IDREF or IDREFS, then insert $value(t, p0)$ in V_{IDREF}. □

Condition C2 holds when V_{ID} has no duplicate and condition C3 is verified when V_{IDREF} only contains values that appear in V_{ID}.

Example 4. We consider $\mathcal{X}$, composed by the schema $\mathcal{D}$ of Example 3, the tree $\mathcal{T}$ depicted in Figure 1 and the run $R = (r, V_{ID}, V_{IDREF})$. The structure $\mathcal{R}$ contains the running tree r (Figure 2) and[2] $V_{ID} = V_{IDREF} = \{99, 00123\}$. As $r(\epsilon) = q_{Shop}$ and $Q_f = \{q_{Shop}\}$, the running tree r is successful. Notice that conditions C1-C3 are respected by $\mathcal{X}$. □

[2] In table Æ of Example 3, the only ID attributes considered are at positions 00 and 10. From Figure 1, $value(t, 000) = \{99\}$ and $value(t, 100) = \{00123\}$.

2.4 Validity of XML Dossiers

Now XML dossiers and their validity are defined according to the preceding sections.

Definition 8. – XML dossier and validity: An XML dossier is a triple $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ where $\mathcal{D}$ is a schema as specified in Definition 5, $\mathcal{T}$ is an XML tree as introduced in Definition 2 and $\mathcal{R}$ is a run obtained according to Definition 7. Two XML dossiers are equal if their components are equal. An empty dossier has $dom(t)$, $dom(r)$, V_{ID} and V_{IDREF} empty.
An XML dossier $\mathcal{X}$ is *valid* if it is empty or if its run $\mathcal{R} = (r, V_{ID}, V_{IDREF})$ respects the following conditions: (i) r is *successful* (Definition 6), (ii) V_{ID} is a *set* and (iii) values in V_{IDREF} exist in V_{ID}. □

We distinguish between two types of validity: the *global* validity of Definition 8 and the *local* one, introduced below. The id-storage V_{IDREF} of a *locally* valid dossier can refer to ID attributes not present in this dossier. This notion is very useful in an update context. For instance, when inserting a locally valid dossier $\mathcal{X}_1$ into a valid dossier $\mathcal{X}$, it is reasonable to suppose that the id-storage V_{IDREF1} contains references to attributes in V_{ID} that are not in V_{ID1}.

Definition 9. – Local validity: Let $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ be an XML dossier where $\mathcal{T} = (t, type, value)$ and $\mathcal{R} = (r, V_{ID}, V_{IDREF})$. Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be the ENFTA in $\mathcal{D}$. An XML dossier $\mathcal{X}$ is locally valid if its run $\mathcal{R}$ respects the following conditions: (i) $r(\epsilon) = q$ and $q \in Q$ and (ii) V_{ID} is a *set.* □

Example 5. The dossier of Example 4 is valid.
Now, we consider the dossier $\mathcal{X}_1 = (\mathcal{D}, \mathcal{T}_1, \mathcal{R}_1)$ with the same schema $\mathcal{D}$ as Example 3. Let the Σ-valued tree t_1 in $\mathcal{T}_1$ be a subtree similar to the one rooted at position 1 of Figure 1 (*i.e.*, having the same labels). The running tree r_1 in R_1 corresponds to the subtree rooted at position 1 in Figure 2. It has $r_1(\epsilon) = q_{Invoice}$, (with $q_{Invoice} \in Q$ and $q_{Invoice} \notin Q_f$), $V_{ID1} = \{00123\}$ and $V_{IDREF1} = \{99\}$. Clearly, $\mathcal{X}_1$ is not valid. However, it is locally valid. □

In the following, we introduce the notion of *sub-dossier* and one important property concerning them.

Definition 10. – Sub-dossier: Let $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ be an XML dossier where $\mathcal{T} = (t, type, value)$ and $\mathcal{R} = (r, V_{ID}, V_{IDREF})$. Let p be a position in $dom(t)$. The XML dossier $\mathcal{X}_p = (\mathcal{D}, \mathcal{T}_p, \mathcal{R}_p)$, where $\mathcal{T}_p = (t_p, type, value)$ and $\mathcal{R}_p = (r_p, V_{IDp}, V_{IDREFp})$, is the sub-dossier of $\mathcal{X}$ at position p if the following conditions hold:

1. $dom(t_p) = \{u \mid pu \in dom(t)\}$
2. $t_p(u) = t(pu)$ for each $pu \in dom(t)$
3. Similarly to t_p, the new functions $type$ and $value$ (associated with $\mathcal{T}_p$) are mappings over $dom(t_p)$ and, therefore, are defined following the same principle as the definition of t_p.
4. The run $\mathcal{R}_p$ is obtained according to Definition 7, from $\mathcal{D}$ and $\mathcal{T}_p$. □

Proposition 1. *If $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ is a valid XML dossier then for every position $p \in dom(t)$, its associated sub-dossier $\mathcal{X}_p = (\mathcal{D}, \mathcal{T}_p, \mathcal{R}_p)$ is locally valid.* □

In Example 5, the dossier $\mathcal{X}_1$ is the sub-dossier of $\mathcal{X}$ at position 1.

3 Updating Valid XML Documents

We define four update operations over XML dossiers showing all changes (on structure and values) that should be performed on their components. Our update processing transforms a valid XML dossier into a (sometimes new) valid XML dossier. Updates that do not preserve validity are rejected. Given a valid dossier $\mathcal{X}$ and a position p, it is possible to update it by performing one of the following operations: $insert(\mathcal{X}_p, p, \mathcal{X})$ (inserts a dossier $\mathcal{X}_p$ in $\mathcal{X}$ at $p \in fr^{ins}(t)$), $insertBefore(\mathcal{X}_p, p, \mathcal{X})$ (inserts $\mathcal{X}_p$ in $\mathcal{X}$ at $p \in dom(t)$), $delete(p, \mathcal{X})$ (deletes from $\mathcal{X}$ the sub-dossier associated to $p \in dom(t)$) and $replace(\mathcal{X}_p, p, \mathcal{X})$ (replaces in $\mathcal{X}$ the sub-dossier associated with p by $\mathcal{X}_p$). Figure 4 displays these operations, showing the changes occurring in a Σ-valued tree t.

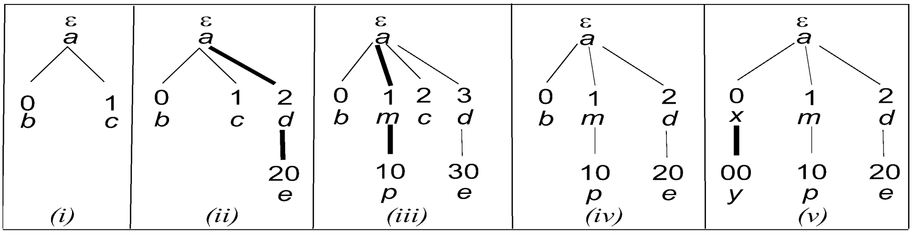


Fig. 4. (i) Initial Σ-valued tree t having labels a (position ϵ), b (position 0) and c (position 1). (ii) Insertion at $p = 2$. (iii) Insertion before $p = 1$. (iv) Deletion at $p = 2$. (v) Replace at $p = 0$.

Next we formally define our set of update operations. Notice that we only consider insertion and deletion of non empty locally valid dossiers.

Definition 11. – Update: Let $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ be a valid dossier. The result of applying an update operation on $\mathcal{X}$ at position p is a valid dossier $\mathcal{X}'$ defined by:

$$\mathcal{X}' = \begin{array}{ll} (\mathcal{D}, \mathcal{T}', \mathcal{R}') & \text{if } (\mathcal{D}, \mathcal{T}', \mathcal{R}') \text{ is a valid} \\ & \text{dossier different from } \mathcal{X}. \\ \mathcal{X} & \text{otherwise.} \end{array}$$

where $\mathcal{T}' = (t', type, value)$ and $\mathcal{R}' = (r', V'_{ID}, V'_{IDREF})$ respect the three properties stated below, which are based on the following assumptions:

- The update position is $p = ui$, with $i \in \mathbb{N}$ and $u \in \mathbb{N}^*$. It is defined according to the update operation: (i) $p \in fr^{ins}(t)$, for *Insertion*,
(ii) $p \in dom(t)$ and $p \neq \epsilon$, for *Insertion before* p and
(iii) $p \in dom(t)$, for *Deletion* and *Replace*.
- $n = |children(father(t, p))| - 1$ for $p \neq \epsilon$.
- $DelPos = \ {}^{k=n}_{k=i}\{w \mid w \in dom(t) \text{ and } w = uku'\}$ if $p \neq \epsilon$, otherwise $DelPos = dom(t)$.
- $ShiftRightPos = \ {}^{k=n}_{k=i}\{w \mid w = u(k+1)u' \text{ and } uku' \in dom(t)\}$.
- $ShiftLeftPos = \ {}^{k=n}_{k=i+1}\{w \mid w = u(k-1)u' \text{ and } uku' \in dom(t)\}$ if $p \neq \epsilon$, otherwise $ShiftLeftPos = \emptyset$.

• $\mathcal{X}_p = (\mathcal{D}, \mathcal{T}_p, \mathcal{R}_p)$ is a non empty locally valid XML dossier with $\mathcal{T}_p = (t_p, type, value)$ and $\mathcal{R}_p = (r_p, V_{IDp}, V_{IDREFp})$.

Properties

1. t' is a Σ-valued tree over $dom(t')$ whose definition depends on the type of update:

(a) *Insertion*: $dom(t') = dom(t) \cup \{pv \mid v \in dom(t_p)\}$ and
$t'(w) = t(w) \quad \forall w \in dom(t)$
$t'(pv) = t_p(v) \; \forall v \in dom(t_p)$
(b) *Insertion before* p: $dom(t') = [dom(t) \setminus DelPos] \cup ShiftRightPos \cup \{pv \mid v \in dom(t_p)\}$ and
• $t'(w) = t(w), \forall w \in dom(t)$ and $w \notin DelPos$
• $t'(u(k+1)u') = t(uku')$ for each $u(k+1)u' \in ShiftRightPos$ where $k \in [i..n]$
• $t'(pv) = t_p(v)$ for each $v \in dom(t_p)$
(c) *Deletion*: $dom(t') = [dom(t) \setminus DelPos] \cup ShiftLeftPos$ and
• $t'(w) = t(w)$ for each $w \in dom(t)$ and $w \notin DelPos$
• $t'(u(k-1)u') = t(uku')$ for each $u(k-1)u' \in ShiftLeftPos$ where $k \in [(i+1)..n]$
(d) *Replace*: $dom(t') = [dom(t) \setminus \{v \mid v \in dom(t) \wedge v = pu'\}] \cup \{pv \mid v \in dom(t_p)\}$ and
$t'(w) = t(w) \quad \forall w \in dom(t)$ and $w \neq pu'$
$t'(pv) = t_p(v) \; \forall v \in dom(t_p)$

2. The Σ-valued tree r' and the new functions $type$ and $value$ are defined following the same principle as t' (property 1 above).
3. The id-storage V'_{ID} and V'_{IDREF} are defined according to the type of update:

(a) *Insertion* and *Insertion before* p:
• V'_{ID} is the set $V_{ID} \cup V_{IDp}$.
• V'_{IDREF} is the bag $V_{IDREF} \cup V_{IDREFp}$ such that every value in V'_{IDREF} exists in V'_{ID}.
(b) *Deletion*: Let $\mathcal{X}_p$ be the sub-dossier of $\mathcal{X}$ at position p.
• V'_{ID} is the set $V_{ID} \setminus V_{IDp}$.
• V'_{IDREF} is the bag $V_{IDREF} \setminus V_{IDREFp}$ such that every value in V'_{IDREF} exists in V'_{ID}.
(c) *Replace at* p: Let $oldp = p$ and let $\mathcal{X}_{oldp} = (\mathcal{D}, \mathcal{T}_{oldp}, \mathcal{R}_{oldp})$, with $\mathcal{R}_{oldp} = (r_{oldp}, V_{IDoldp}, V_{IDREFoldp})$, be the sub-dossier of $\mathcal{X}$, at position *oldp*, to be replaced by the new dossier $\mathcal{X}_p = (\mathcal{D}, \mathcal{T}_p, \mathcal{R}_p)$.
• V'_{ID} is the set $(V_{ID} \setminus V_{IDoldp}) \cup V_{IDp}$.
• V'_{IDREF} is the bag $(V_{IDREF} \setminus V_{IDREFoldp}) \cup V_{IDREFp}$ such that every value in V'_{IDREF} exists in V'_{ID}. □

The following theorem states that when the resulting dossier is different from the original one, the update operation has *effectively* been performed.

Theorem 1. *Let $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ be a valid dossier and let $\mathcal{X}_p = (\mathcal{D}, \mathcal{T}_p, \mathcal{R}_p)$ be a locally valid dossier. Let $\mathcal{X}' = (\mathcal{D}, \mathcal{T}', \mathcal{R}')$ be a dossier* different*from $\mathcal{X}$ and let p be a position.*

• *If* $\mathcal{X}' = insert(\mathcal{X}_p, p, \mathcal{X})$ *and* $p \in fr^{ins}$ *then* $\mathcal{X}'$ *is valid and* $\mathcal{X}_p$ *is the sub-dossier of* $\mathcal{X}'$ *at* p*.*
• *If* $\mathcal{X}' = insertBefore(\mathcal{X}_p, p, \mathcal{X})$ *then* $\mathcal{X}'$ *is valid and* $\mathcal{X}_p$ *is the sub-dossier of* $\mathcal{X}'$ *at* p*. Each sub-dossier of* $\mathcal{X}$ *at* p *and its right siblings is a sub-dossier of* $\mathcal{X}'$*, shifted one position to the right.*
• *If* $\mathcal{X}' = delete(p, \mathcal{X})$ *then* $\mathcal{X}'$ *is valid and* $\mathcal{X}_p$ *is the sub-dossier of* $\mathcal{X}$ *at* p*, but it is not the sub-dossier of* $\mathcal{X}'$ *at* p*. Each sub-dossier of* $\mathcal{X}$ *at a position that is a right sibling of* p *is a sub-dossier of* $\mathcal{X}'$*, shifted one position to the left.*
• *If* $\mathcal{X}' = replace(\mathcal{X}_p, p, \mathcal{X})$ *then* $\mathcal{X}'$ *is valid and* $\mathcal{X}_p$ *is the sub-dossier of* $\mathcal{X}'$ *at* p *and there exists* $\mathcal{X}_{oldp}$ *which is the sub-dossier of* $\mathcal{X}$ *at* p*.* □

Proof (Sketch): From Definition 11, it can be verified for each update operation that the validity conditions (Definition 8) hold.

We finish this section by stating the correction and the completeness of our update operators. In other words, we show that after an update, a valid XML dossier remains valid and that any valid dossier can be obtained from a sequence of our update operations.

Lemma 1. *Let* $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ *be a valid dossier. The XML dossier* $\mathcal{X}'$ *resulting from the update of* $\mathcal{X}$ *according to Definition 11 is valid.* □

Theorem 2. *Let* $\mathcal{X}$ *and* $\mathcal{X}'$ *be valid dossiers with respect to a schema* $\mathcal{D}$*. There exists a sequence* u *of update operations (of Definition 11) such that* $\mathcal{X}'$ *is the result of applying* u *over* $\mathcal{X}$*.* □

Proof: The proof is straightforward since $\mathcal{X}' = replace(\mathcal{X}', \epsilon, \mathcal{X})$.

4 Incremental Validation

In this section we explain how to perform *incremental* validity tests before accepting an update. From Definition 11, we notice that an update over $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ at position p only results in changes to p's right siblings (including itself). Thus, only the subtree rooted at $father(p)$ has to be checked to assure the validity of the updated document.

All update procedures have the dossier $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$ and the update position p as input. Procedures $insert$, $insertBefore$ and $replace$ receive a dossier $\mathcal{X}_p$=($\mathcal{D}$, $\mathcal{T}_p$, $\mathcal{R}_p$), to be added to $\mathcal{X}$. We consider that $\mathcal{X}$ is valid, p is a correct update position and $\mathcal{X}_p$ is locally valid (or valid, according to the type of update). These assumptions can be easily verified at the beginning of update procedures.

To implement incremental validity tests efficiently, we *simulate* the update using a small auxiliary run $\mathcal{R}_{aux} = (r_{aux}, ID_{aux}, IDREF_{aux})$. The id-storage ID_{aux} and $IDREF_{aux}$ are *bags*. They are computed according to the type of update, implementing the operations used in the Property 3 of Definition 11. The Σ-valued tree r_{aux} is always an 1-depth tree. As shown below, and illustrated by Figure 5, r_{aux} is built with the siblings of p in r, and by re-computing the state of p's father.

Construction of r_{aux}:
Leaves: Let $p = ui$ be the update position, with $u \in \mathbb{N}^*$ and $i \in \mathbb{N}$:

1. Copy the left siblings of position p:
   For $j \in [0..i-1]$ do $r_{aux}(j) = r(uj)$
2. Let[3] $n = |children(r, father(r,p))| - 1$ and compute the other leaves according to the update operation:

   For $insert$: $r_{aux}(i)$=$r_p(\epsilon)$. In this case, i is the rightest child.
   For $insertBefore$:
   $r_{aux}(i)$=$r_p(\epsilon)$
   for $k \in [i..n]$ do $r_{aux}(k+1) = r(uk)$
   For $delete$:
   for $k \in [i..(n-1)]$ do $r_{aux}(k) = r(u(k+1))$
   For $replace$:
   $r_{aux}(i)$=$r_p(\epsilon)$
   for $k \in [(i+1)..n]$ do $r_{aux}(k) = r(uk)$

Root: Compute the root $r_{aux}(\epsilon)$ by applying the transition rule associated with the label $t(father(t,p))$.

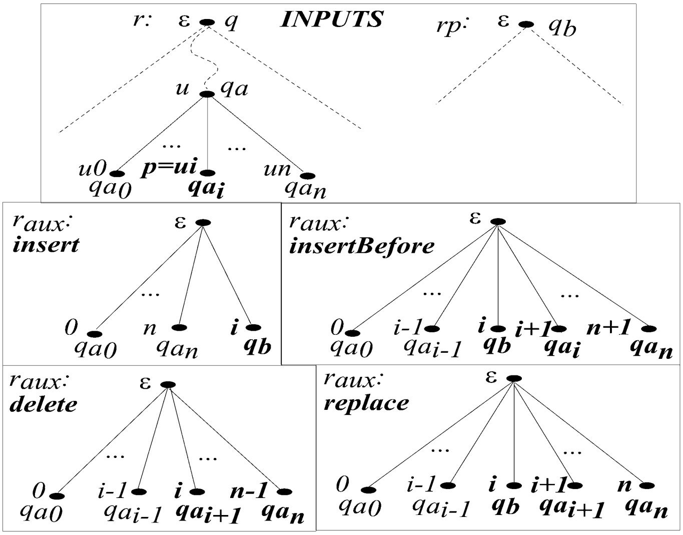


Fig. 5. Auxiliary trees r_{aux} for each type of update.

[3] We recall that $dom(t) = dom(r)$ and that functions $children$ and $father$ return positions in $dom(r)$ (or $dom(t)$). Thus, $|children(r, father(r,p))|$ gives the number of children of p's father.

Once $\mathcal{R}_{aux}$ is built, the validity tests are performed by checking
(i) if $r(father(t,p))$ equals $r_{aux}(\epsilon)$ and
(ii) if ID_{aux} and $IDREF_{aux}$ respect the validity conditions stated in Definition 8.

Notice that $r_{aux}(\epsilon)$ represents the state that should be associated with position $father(t,p)$ if the update was accepted. In fact, when we test if $r(father(t,p))$ equals $r_{aux}(\epsilon)$, we are taking into account the good properties of our tree automaton $\mathcal{A}$. Since $\mathcal{A}$ is a translation of an unambiguous DTD [9], each label $a \in \Sigma$ is associated with a *unique* state q_a. Thus, both running trees r (before an update) and r' (after an accepted update) should associate the same state with position $father(t,p)$. To verify if an update respects this property, we perform the test on r_{aux} (instead of building the whole r').

Now we make some remarks concerning complexity. Given a dossier $\mathcal{X} = (\mathcal{D}, \mathcal{T}, \mathcal{R})$, the construction of $\mathcal{R}$ from $\mathcal{D}$ and $\mathcal{T}$ is linear in the number of nodes appearing in the Σ-valued tree t (in $\mathcal{T}$). To visit a subtree of t in order to fill bags containing ID values is linear in the number of nodes of the subtree. The construction of ID_{aux} and $IDREF_{aux}$, from the original bags V_{ID}, V_{IDREF}, etc., is linear in the number of ID/IDREF(S) values present in these original bags. Considering m as the maximum number of values in ID_{aux} and $IDREF_{aux}$ checking the conditions (ii) and (iii) of Definition 8 takes, in the worst case, time $O(m^2)$. The construction of $\mathcal{R}_{aux}$ takes linear time in the number of p's siblings. Applying the transition rule over $\mathcal{R}_{aux}$ also takes linear time in the number of p's siblings (*i.e.*, positions in $dom(r_{aux})$).

Therefore, in comparison with a naive method that first applies the update and then checks validity from scratch, it is easy to see that our approach is far better. Supposing that the input of the update operations respect the assumptions of Definition 11, we just need to construct some small auxiliary structures and to apply a transition rule *once*. In general the gain is very important: imagine for example that we delete, insert or replace a “phone number” of an element “person” in a large document. The incremental validity test will check only the element “person” concerned by the update, and not the whole resulting document[4]! Not counting that, in the naive method, if the resulting document is not valid then all update process must be rolled back.

5 Conclusions

Tree automata are used in XML research in different ways (see [15,18] as surveys). For instance, several static typing techniques for XML transformers (including static type checking [4,14] and type inference [11,16]) have been modeled with tree automata or tree transducers. In [14], the authors consider the type checking problem expressed by k-pebble transducers, showing that it is decidable. In [4], they consider trees with labels from an infinite alphabet in order to represent both elements and their values,

[4] As another example, consider that an element “person” is added in a long list of “person” elements (originally valid): the incremental validity test will check only the local validity of this new element and will accept the update if this element is locally valid. This is possible because adding an element in a list is obviously correct when this list of sub-elements is specified in the schema (and it is, since the original list is valid).

showing that in this case type checking becomes undecidable. The problem of type check transformations of XML trees (given a tree, its schema and a transformation, check whether the resulting tree conforms to a specified schema) is complementary to the one we address in this paper, as it focus on the task of extracting (sometimes with restructuring) information from a given document.

In this paper we have applied unranked tree automata to the *incremental validation of updates*. Our validity test is static, as we perform it before applying the update. For this purpose, we use the extended tree automaton introduced in [5] whose aim is to deal with both element and attribute constraints. This automaton has the same expression power as a DTD and gives rise to an efficient validation method. It has already been implemented as a validator from scratch using both SAX and DOM. The key proposition to perform incremental validation efficiently is to build a temporary sequence of states which represents the requested change (together with bags of ID/IDREF(S) values) and to perform tests upon this small auxiliary structure.

Updates and incremental validation are very useful for many application such as XML databases and XML editors. To our knowledge no information is provided on the incremental validation of updates for the available products on XML [17,1].

A set of primitive XML update operations (different from ours) is proposed in [20]. Contrary to us, the authors are not interested in the problem of validation. Their goal is to define an XML update language and to translate update operations into updates on the associated relational database. The same problem is addressed in [7] where, in order to specify when XML views are updatable, the authors use the nested relational algebra as the formalism for defining them. As XML views must respect the schema induced by the view specification, in [7] only updates that do not violate it are considered. However, contrary to us, the goal of the authors is not to build an update environment that assures the validity of XML views. In this sense, our work is complementary to theirs.

Our choice of update operations is based on existing propositions for extending XQuery with updates. For instance, [13] enumerates studies of XML update operations and conduct experimental study to compare their incremental checking method against re-validating the whole document after an update. Note that they use a quite different approach to pre-validate updates, based on constraint check queries, which seems to add unnecessary computation overhead. As in [13,17], we also have implemented a *rename* operation, which gives a new name to a node (element or attribute), not presented here because it is just a simplified version of a *replace*.

Updates on trees appear in papers, such as [19], dealing with the notion of distance between two trees. Their update operations are general while ours reflect desired changes on XML documents. The notion of distance between two trees is also explored in [12], where the goal is to detect changes and not to propose updates that allow such changes.

In [17] the authors propose an incremental validation of XML documents. They first describe a way to incrementally check the sequence composed by sub-element states. They maintain a kind of B-tree as auxiliary information structure. This idea is extended to incremental DTD validation in the case of *one element renaming*. Finally, for specialized DTDs they propose to use as auxiliary structure a binary tree encoding of the document. This structure is of size $O(n)$, where n is the size of the document, and their incremental validation is in time $O(log^2 n)$.

Although the aim of the work presented in [17] is similar to ours, the two proposals differ in many aspects. Contrary to [17], we deal not only with element constraints but also with attribute constraints. In [17], only elementary updates affecting *one* node at a time are considered. Our update method allows sophisticated update operations (dealing with trees) without loosing the capacity of effectively performing elementary ones.

In terms of complexity, in our approach, considering only the insertion or deletion of a leaf (at position p) in the XML tree (t being the Σ-valued tree), we improve the complexity bounds. Our auxiliary structure has size $|dom(r_{aux})|$ and our validation time is linear in the number of elements in $dom(r_{aux})$. In [17] the same operation takes time $O(log^2\ n)$ where n is the size of the entire document, *i.e.*, n equals $|dom(t)|$. However, contrary to them, we do not consider specialized DTDs.

We are currently considering the following lines of research:

(i) The construction of a framework for manipulating XML documents. This framework is intended to be a formal laboratory to test query and update languages for XML [2,3]. We are currently implementing our update operations using the ASF+SDF [8] meta-environment. Next, we shall consider the development of an XML update language as an extension of some existing query language such as XQuery. To this end, we shall define a method to determine the update position p by the evaluation of some predicates (*e.g.*, XPath). In doing this, we intend also to investigate how the intermediate validation necessary to determine p can optimize the complete validation for the data modification operation.

(ii) The generalization of our method to treat other kinds of updates such as those that change the schema since they can help a lot the administration of a data exchange environment.

(iii) The generalization of the update process to consider ”global” updates, *i.e.*, a sequence (or a set) of updates treated as one unique transaction, instead of just a single primitive update operation. In this case, we are interested in assuring validity just after considering the whole sequence of updates - and not after each update of the sequence, independently. In other words, as a valid document is transformed using a sequence of primitive operations, the document can be temporarily invalid but in the end validity is restored.

(iv) The extension of our method to deal with specialized DTDs, as well as to treat integrity constraints. Our goal is to incrementally validate updates over XML documents, taking into account both schema and key constraints (as defined in [10]). To this end, we aim at merging the method presented here with the proposal in [6]. This first extension is our way to start the investigation of how our validation process can work when XML Schema (instead of DTD) is considered.

References

1. XML editor products. Available at http://www.perferctxml.com/soft.asp?cat=6.
2. XML query working group. Available at http://www.w3.org/XML/Query.
3. XUpdate - XML:DB Working draft. Available at http://www.xmldb.org/xupdate/xupdate-wd.html.
4. N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *ACM Symposium on Principles of Database System*, 2001.
5. B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, and D. Laurent. Extending tree automata to model XML validation under element and attribute constraints. In *ICEIS*, 2003.
6. B. Bouchou, M. Halfeld Ferrari Alves, and M. A. Musicante. Tree automata to verify key constraints. In *Web and Databases (WebDB)*, San Diego, CA, USA, June 2003.
7. V. P. Braganholo, S. B. Davidson, and C. A. Heuser. On the updatability of XML views over relational databases. In *Web and Databases (WebDB)*, San Diego, CA, USA, June 2003.
8. M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling rewrite systems: The ASF+SDF compiler. *ACM, Transactions on Programming Languages and Systems*, 24, 2002.
9. A. Brüggeman-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.
10. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. C. Tan. Keys for XML. In *WWW10, May 2-5*, 2001.
11. B. Chidlovskii. Using regular tree automata as XML schemas. In *Proc. IEEE Advances in Digital Libraries Conference*, May 2000.
12. G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *Data Engineering*, 2002.
13. B. Kane, H. Su, and E. A. Rundensteiner. Consistently updating XML documents using incremental constraint check queries. In *Proceedings of WIDM 02*, 2002.
14. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *ACM Symposium on Principles of Database System*, pages 11–22, 2000.
15. F. Neven. Automata, logic and XML. In *CSL'02 - Annual Conference of the European Association for Computer Science Logic (invited talk)*, 2002.
16. Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *ACM Symposium on Principles of Database System*, pages 35–46, 2000.
17. Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2003.
18. D. Suciu. On database theory and XML. *SIGMOD Record*, 30(3), 2001.
19. Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery*, 26(3), 1979.
20. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *ACM SIGMOD*. ACM, 2001.
21. W. Thomas. Automata of infinite objects. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.

Attribute Grammars for Scalable Query Processing on XML Streams

Christoph Koch[1,⋆] and Stefanie Scherzinger[2]

[1] LFCS, University of Edinburgh, Edinburgh EH9 3JZ, UK
koch@dbai.tuwien.ac.at
[2] Lehrstuhl für Dialogorientierte Systeme
FMI, Universität Passau, D-94030 Passau, Germany
scherzin@fmi.uni-passau.de

Abstract. We introduce the new notion of XML Stream Attribute Grammars (XSAGs). XSAGs are the first scalable query language for XML streams (running strictly in linear time with bounded memory consumption independent of the size of the stream) that allows for actual data transformations rather than just document filtering. XSAGs are also relatively easy to use for humans. Moreover, the XSAG formalism provides a strong intuition for which queries can or cannot be processed scalably on streams. We introduce XSAGs together with the necessary language-theoretic machinery, study their theoretical properties such as their expressiveness and complexity, and discuss their implementation.

1 Introduction

In recent years, XML has become a standard format for document exchange and now seems to develop into a preeminent representation language for streaming data as well. This development calls for flexible query languages for processing streams which support data transformations.

In [12, 15, 7], fragments of the standard XML Query language [16] are evaluated on XML streams. These fragments tend to support powerful data transformations, with the consequence that query processing neither scales in terms of runtime nor memory consumption. Indeed, in these works, memory buffers are required that can grow arbitrarily large, depending on the amount of data communicated via the stream.

This problem is due to the nature of XML Query, which renders it ill-suited for stream processing: Features such as nested for-loops with transitive paths (e.g., using the descendant axis), which may lead to a nonlinearly-sized output, and nonlocal computations such as joins and the reordering and sorting of data cannot be handled scalably on streams. In addition, the syntax of XML Query makes it difficult to tell for a user whether a query can – at least in principle – be evaluated scalably, in linear time using little memory.

[⋆] Work sponsored by Erwin Schrödinger Grant J2169 of the Austrian Research Fund (FWF).

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 233–256, 2004.

Query languages that require unbounded memory buffers constitute a scalability issue on streams and are not in the spirit of the database community's quest for tailored formalisms that provide the appropriate tradeoffs between expressiveness and complexity for the data management challenge at hand.

XML streams by definition may be *very* long, or should even be assumed to be infinite. For query processing to be feasible on streams, there is a need for special-purpose query languages and evaluation algorithms which *scale to streams*, i.e.,

(a) which can be evaluated strictly in linear time in the size of the input,
(b) which work in streaming fashion, by one linear forward scan of the data, and
(c) for which, at any time during query evaluation, memory consumption is bounded[1].

Among the models of computation that allow for better control of complexity than languages such as XML Query, there are various forms of automata/transducers and certain attribute grammars. The former are, however, unsuitable as query languages used by humans because their specifications tend to be large, technical, and hard to read. The latter approach is developed in the present paper.

Attribute Grammars for Stream Processing

In this work, we develop and investigate a formalism for processing XML streams called *XML Stream Attribute Grammars* (XSAGs), a new class of attribute grammars specifically designed for scalable XML stream processing. XSAGs can be evaluated strictly in linear time in streaming fashion, consuming only a stack of memory bounded by the depth of the XML tree being streamed. Thus, XSAGs satisfy our desiderata (a) through (c).

XSAGs are based on *extended regular tree grammars*, i.e. regular tree grammars in which the right-hand sides of productions may contain regular expressions, allowing to specify nodes in the parse tree that have an unbounded number of children. Extended regular tree grammars are thus well-suited for specifying classes of unranked trees denoting XML documents. We assume that extended regular tree grammars are often available for XML streams in the dialect of *Document Type Definitions* (DTDs). This adds to the relevance of the present formalism.

An XSAG is obtained by annotating a given extended regular tree grammar with attribution functions that describe the output to be produced from the input stream. In the tradition of L-attributed grammars [1], XSAGs are assumed

[1] Note that a stack of memory proportional to the maximum depth of the XML tree is necessary for even the most basic sequential navigation and parsing tasks (see e.g. [8, 10]). To be precise, in (c) we thus call for memory consumption that is bounded w.r.t. the length of the stream but not the depth of the XML tree (an indication of its structural complexity). XML trees tend to be very shallow but wide, therefore such a stack is not considered a bottleneck to scalability.

to perform a single scan of the XML stream, which amounts to effecting a single depth-first left-to-right traversal of the document tree. Also in the tradition of L-attributed grammars, right-hand sides of productions can be annotated with *two* attribution functions, one – placed at the beginning of the right-hand side – that is executed upon reaching the opening tag of a node in the XML document (or equivalently, when descending into a subtree), and the second – placed at the end of the right-hand side – which is executed when the corresponding closing tag is reached (or, equivalently, when returning from the depth-first-traversal of the subtree).

Example 1. Consider the extended regular tree grammar $G = (Nt, T, P, bib)$ with nonterminals

$$Nt = \{bib, book, article, title, author\},$$

start nonterminal *bib*, terminals

$$T = \{\mathrm{bib}, \mathrm{book}, \mathrm{article}, \mathrm{title}, \mathrm{author}, PCDATA\},$$

and the productions P

$$\begin{aligned} bib &::= \mathrm{bib}((book \cup article)^*) \\ book &::= \mathrm{book}(title.author.author^*) \\ article &::= \mathrm{article}(title.author.author^*) \\ title &::= \mathrm{title}(PCDATA) \\ author &::= \mathrm{author}(PCDATA) \end{aligned}$$

which defines an XML bibliography database.

By changing the first production to

$$bib ::= \{ECHO\}\ \mathrm{bib}((book \cup article)^*)$$

we obtain an XSAG that simply echoes the input stream. Indeed, the start production matches the root node of the document, and *ECHO* writes the entire subtree of the current node to the output as XML.

If we are instead only interested in books arriving on the stream, we can use the XSAG obtained by changing the *bib* and *book* productions to

$$\begin{aligned} bib &::= \{\mathrm{print}\ \langle\mathrm{books}\rangle\}\ \mathrm{bib}\ \ (book \cup article)^* \quad \{\mathrm{print}\ \langle/\mathrm{books}\rangle\} \\ book &::= \{ECHO\}\ \mathrm{book}\ \ title.author.author^* \end{aligned}$$

Here we apply *ECHO* to the *book* subtrees, but not to articles. We explicitly output the opening and closing tags of the root node, and label the root node of the output produced by this XSAG “books”, rather than “bib”. □

Based on the basic notion of XSAGs (bXSAGs) exemplified so far, we introduce the *easy XSAGs* (yXSAGs). These allow to annotate the regular expressions inside productions with attribution functions as well, which adds to the flexibility of the formalism.

Example 2. The yXSAG with production

$$\begin{array}{lll}
\mathit{article} ::= & \{\text{print } \langle\text{article}\rangle\} & \\
& \text{article } (\{\mathit{ECHO}\}\mathit{title}). & \\
& & (\{\text{print } \langle\text{authors}\rangle;\ \mathit{ECHO}\} \\
& & \quad (\mathit{author}.\mathit{author}^{*}) \\
& & \{\text{print } \langle/\text{authors}\rangle\}) \\
& \{\text{print } \langle/\text{article}\rangle\} &
\end{array}$$

basically outputs articles as they arrive on the stream, but groups the authors of each article under a common *authors* node. Here, the second appearance of *ECHO* in the production applies to the tree region matched by the regular expression $\mathit{author}.\mathit{author}^{*}$, i.e., to the subtrees below *article* nodes that are rooted by *author* nodes. □

Being attribute grammars, XSAGs of course support attributes. In order to assure scalability in the strictest sense, we require that attributes range over a finite domain fixed with the XSAG.

Example 3. Assume that our grammar assures that books in addition have a year of publication as a first child:

$$\begin{aligned}
\mathit{book} &::= \text{book } \ \mathit{year}.\mathit{title}.\mathit{author}.\mathit{author}^{*} \\
\mathit{year} &::= \text{year}(\mathit{PCDATA})
\end{aligned}$$

Then, for instance, the yXSAG

$$\begin{array}{ll}
\mathit{bib} ::= & \{\text{print } \langle\text{books}\rangle\} \text{ bib } (\mathit{book} \cup \mathit{article})^{*} \quad \{\text{print } \langle/\text{books}\rangle\} \\
\mathit{book} ::= & \text{book } (\{\mathit{MATCH\_CHILDREN}(\text{“2003”}, c)\}\ \mathit{year}). \\
& \quad (\{\text{if } c = \mathit{true} \text{ then} \\
& \qquad \text{begin} \\
& \qquad\quad \text{print } \langle\text{book}\rangle; \mathit{ECHO} \\
& \qquad \text{end}\} \\
& \qquad (\mathit{title}.\mathit{author}.\mathit{author}^{*}) \\
& \quad \{\text{if } c = \mathit{true} \text{ then print } \langle/\text{book}\rangle\})
\end{array}$$

outputs books whose year of publication is “2003” with their *title* and *author* children, but without the *years*.

For a given *year* node, *MATCH_CHILDREN* sets boolean-valued condition attribute[2] c to true if the character text encountered while scanning its children from left to right matches “2003”. Otherwise, c is set to false.

This condition attribute is passed on through the document tree during its traversal. The regular expression $\mathit{title}.\mathit{author}.\mathit{author}^{*}$ describes a tree region among the children of a *book* node. Just before we first enter this tree region,

[2] Note that in the technical sections of this paper, we will use a somewhat more explicit syntax when employing attributes (see e.g. Example 14).

we examine the value of c. If c is true (and the current book has been published in 2003), we print opening tag ⟨book⟩ and echo the tree region to the output. In this case, we further output closing tag ⟨/book⟩ on leaving the tree region.

Note that this yXSAG is equivalent to XML Query

```
⟨books⟩
{ for $x in //book where $x/year = 2003
  return ⟨book⟩ {$x/title} {$x/author} ⟨/book⟩ }
⟨/books⟩
```

on documents conforming to our grammar. □

Attribute grammars are well known in the field of compilers and have recently been revisited in the context of XML, for instance for grammar-directed XML publishing [4, 3]. Some of their theory relevant in the context of structured documents has been studied in [14, 13].

Our emphasis is on designing a *practical* formalism for query processing that is *relatively easy to use*. Attribute grammars are widely agreed to carry a strong intuition for specifying syntax-directed translations. In our setting, they provide a metaphor for strictly linear-time one-pass XML transformations that can be grasped very intuitively. This renders it relatively easy for a user to recognize or design queries which can be executed (scalably) on a stream, even if this intuition is paid for by our formalism being more operational than languages such as XML Query. While ease of use cannot be conclusively asserted based only on our own observations and the examples we provide, alternative formalisms such as deterministic pushdown transducers (DPDTs) are unsuitable as query languages to be used by humans; query processors for languages such as XML Query, on the other hand, do not scale to streams. We can therefore argue that XSAGs achieve our goal of relative ease of use. Already bXSAGs are much more practical than DPDTs. yXSAGs permit elegant nested attributions, which, as can be seen in Example 2 and others throughout the paper, allow to specify many interesting data transformations conveniently.

Contributions

The technical contributions of this paper are as follows.

- We examine the framework of extended regular tree grammars and craft grammar classes appropriate for attribution and stream processing.
- In order to be able to characterize yXSAGs properly, we develop the new notion of *strongly one-unambiguous regular expressions*, as well as some of its theory. These expressions are precisely those for which the *parse tree* of a word (analogously to the derivation tree of a grammar) can be unambiguously constructed *online*, with just a one-symbol lookahead, while processing the stream.

  yXSAGs allow for attributions to be nested inside regular expressions by only permitting strongly one-unambiguous regular expressions in the right-hand sides of productions.

- We introduce and formally define our two notions of attribute grammars, bXSAGs and yXSAGs, and compare them with respect to usability.
- We introduce *XML-DPDT*s, deterministic push-down transducers with a natural stack discipline that assures that the size of the stack remains strictly proportional to the depth of the XML tree and which can only accept well-formed XML documents. *XML-DPDT*s in a sense capture the intuition of scalable XML stream processing and serve as an expressiveness yardstick for XSAGs.
- We show that both bXSAGs and yXSAGs are precisely as expressive as *XML-DPDT*s. XSAGs provide the same quasi-optimal trade-off between expressiveness and evaluation cost as do *XML-DPDT*s.
- Finally, we study the complexity of evaluating XSAGs, and their implementation.

The *structure* of this paper basically follows the order of contributions described above.

2 Regular Tree Grammars

Throughout this paper, we assume that regular expressions are constructed from atomic symbols using concatenation ., union $\cup$, and the Kleene star * (but not ϵ, $^+$, or ?).

Let *Tag* be a set of node labels ("tags") and let *Char* be a set of characters distinct from the tags. An *extended regular tree grammar* is a grammar $G = (Nt, T, P, s)$ where

1. Nt is a set of nonterminals,
2. $T = Tag \cup Char$ is a set of terminals,
3. P is a set of productions $nt ::= t(\rho)$ where $nt \in Nt$, $t \in T$, ρ is either ϵ or a regular expression over alphabet Nt, and if $t \in Char$, then $\rho = \epsilon$, and
4. $s \in Nt$ is the start symbol.

We assume the standard meaning of grammars and their derivations for which we refer to [9] for basic and to [11] for extended grammars.

Extended regular tree grammars (and DTDs, which are a dialect of extended regular tree grammars) are a convenient way to specify a class of unranked labeled trees and thus XML documents.

Let $Char = \{c_1, \ldots, c_n\}$. As a shortcut, we define the regular expression macro

$$PCDATA := (c'_1 \cup \cdots \cup c'_n)^*$$

which, using new nonterminals $c'_1, \ldots, c'_n$ and productions $c'_i ::= c_i(\epsilon)$ for each $1 \leq i \leq n$, can be used just like a terminal in right-hand sides of grammar productions. PCDATA accepts all character strings. As a further notational convenience, we will allow ourselves to be somewhat imprecise below whenever we only use PCDATA, but no characters, in our grammar definitions. Then we will list "PCDATA" among the terminals and will keep the nonterminals $c'_1, \ldots, c'_n$ unmentioned.

Given a nonterminal *nt*, let $\theta(nt)$ denote the set of terminals t such that the grammar contains a production $nt ::= t(\rho_{nt,t})$. Given a regular expression ρ, let $\tau(\rho)$ denote the regular expression in which each nonterminal *nt* in ρ is replaced by the union of terminals $\quad \theta(nt)$.

Each extended regular tree grammar can be alternatively considered as an extended context-free word grammar (CFG), which is obtained by simply rewriting each right-hand side $tag(\rho)$ into $\langle tag \rangle \rho \langle /tag \rangle$. *Deterministic* context-free languages are precisely those recognizable by the deterministic pushdown automata (DPDA, see e.g. [9]). DPDAs run comfortably on streams requiring only a stack of memory bounded by the depth of the input tree. Using automata, we can thus scalably recognize the deterministic context-free languages.

The problem of processing an (extended) attribute grammar on a document requires an additional, different restriction on the grammar besides determinism to allow for deterministic computation. We need to unambiguously refer to the atomic *symbols* in the regular expressions to be able to access or assign attributes. In attribute grammars, a straightforward solution [13] is to require for right-hand side regular expressions ρ that $\tau(\rho)$ be *unambiguous*.

Example 4. Consider the grammar

$$\begin{aligned} bib &::= \mathrm{bib}((book_1 \cup book_2)^*) \\ book_1 &::= \mathrm{book}(\rho) \\ book_2 &::= \mathrm{book}(\rho) \end{aligned}$$

where ρ is some regular expression. The regular expression $(book_1 \cup book_2)^*$ is unambiguous, but

$$\tau(book_1 \cup book_2)^* = (\mathrm{book} \cup \mathrm{book})^*$$

is not. Therefore, when considering the tags of the children of the (root) *bib* node, we cannot determine where to apply which of the two *book* productions with their possibly different attributions. □

On streams, we cannot look ahead beyond a nonterminal (which may stand for a large subtree that we do not want to buffer) when parsing the input. Thus, we will assume the stronger notion of *one-unambiguity* for regular expressions $\tau(\rho)$. That is, we will require that $\tau(\rho)$ can be unambiguously parsed with just one symbol of lookahead.

2.1 One-Unambiguity and TDLL(1)

By a *marking* of a regular expression ρ over alphabet Σ, we denote a regular expression ρ' such that each occurrence of an atomic symbol in ρ is replaced by the symbol with its position among the atomic symbols of ρ added as subscript. That is, the i-th occurrence of a symbol $a \in \Sigma$ in ρ is replaced by a_i. For instance, the marking of $(a \cup b)^*.a.a^*$ is $(a_1 \cup b_2)^*.a_3.a_4^*$. The reverse of a marking (indicated by $^\#$) is obtained by dropping the subscripts.

Let ρ be a regular expression, ρ' its marking, and Σ' the marked alphabet used by ρ'. Then, ρ is called *one-ambiguous* iff there are words u, v, w over Σ' and symbols $x, y \in \Sigma'$ such that $uxv, uyw \in L(\rho')$, $x \neq y$, and $x^\# = y^\#$. A regular expression is called *one-unambiguous* if it is not one-ambiguous.

Example 5. Consider the regular expression $\rho = a^*.a$ and its marking $\rho' = a_1^*.a_2$. Let $u = a_1$, $x = a_2$, $y = a_1$, $v = \epsilon$, and $w = a_2$. Clearly, $uxv = a_1.a_2$ and $uyw = a_1.a_1.a_2$ are both words of $L(\rho')$, thus ρ is one-ambiguous. On the other hand, the equivalent regular expression $a.a^*$ is one-unambiguous. □

Definition 1 ([11]). A *TDLL(1)-Grammar* is an extended regular tree grammar where $\tau(s)$ is unambiguous[3] and in which for each regular expression ρ in the right-hand side of a production, $\tau(\rho)$ is one-unambiguous. □

Example 6. The grammar of Example 1 is TDLL(1). On the other hand, since the grammar of Example 4 contains a regular expression ρ such that $\tau(\rho)$ is not even unambiguous, that grammar is not TDLL(1). □

Remark 1 (One-unambiguity in DTDs). For XML elements that exclusively have elements as children (but no character data), the W3C recommendation[4] explicitly requires a one-unambiguous *content model* (that is, right-hand side regular expression) in order to assure compatibility with SGML.

Productions defining elements also containing character data must be constructed according to either the pattern

$$nt_0 ::= (\text{PCDATA} \cup nt_1 \cup \cdots \cup nt_m)^*$$

or $nt_0 ::= \text{PCDATA}$ (where $nt_0, \ldots, nt_m$ are DTD element names, i.e., nonterminals). Clearly, all regular expressions such constructed are also one-unambiguous.

Thus, since DTDs also contain at most one production $nt ::= t(\rho)$ for each "element" t (and thus if ρ is one-unambiguous, $\tau(\rho)$ is as well), DTDs are TDLL(1) grammars (see also [11]). □

2.2 Strong One-Unambiguity and STDLL(1)

One-unambiguity and TDLL(1) grammars allow us to use attributed regular tree grammars on XML streams. However, as we show below, the ability to use attribution functions inside the regular expressions at the right-hand sides of productions will allow us to write many practical queries in a much more user-friendly fashion. Our machinery for achieving this is the notion of *strongly one-unambiguous* regular expressions.

[3] Note that $\tau(s)$ is guaranteed to be a very simple form of regular expression, a disjunction of atomic symbols (so unambiguity implies one-unambiguity).

[4] Sections 3.2.1, 3.2.2, and Appendix E in [5].

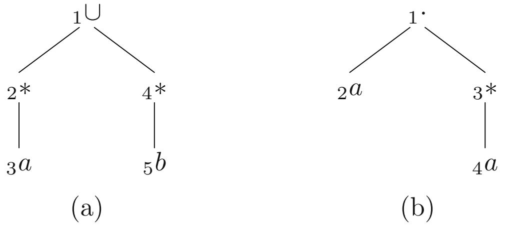


Fig. 1. Parse trees of regular expressions $a^* \cup b^*$ (a) and $a.a^*$ (b) with nodes annotated with markings.

Intuitively, by a *bracketing* of a regular expression ρ, we refer to a *marking of the nodes in the parse tree* of ρ using distinct indexes. We realize this by assigning the indexes by a depth-first left-to-right traversal of the parse tree, that is, in document order (see Figure 1 for two examples). The bracketing $\rho^{[]}$ is then obtained by inductively mapping each subexpression π of ρ with index i to $[_i.\pi.]_i$. Thus, a bracketing is a regular expression over the alphabet $\Sigma \cup \Gamma$, where $\Gamma = \{[_i,]_i \mid i \in \{1, 2, 3, \dots\}\}$. (We assume Σ and Γ disjoint.) For example, $[_1.\ ([_2.([_3.a.]_3)^*.]_2) \cup ([_4.([_5.b.]_5)^*.]_4)\ .]_1$ is the bracketing of $a^* \cup b^*$ and $[_1.\ ([_2.a.]_2).([_3.([_4.a.]_4)^*.]_3)\ .]_1$ is the bracketing of $a.a^*$.

Definition 2. Let ρ be a regular expression and $\rho^{[]}$ be its bracketing. A regular expression ρ is called *strongly one-unambiguous* iff there do not exist words u, v, w over $\Sigma \cup \Gamma$, words $\alpha \neq \beta$ over Γ, and a symbol $x \in \Sigma$ such that $u\alpha xv, u\beta xw \in L(\rho^{[]})$ or $u\alpha, u\beta \in L(\rho^{[]})$. □

Example 7. The regular expression $a^* \cup b^*$ (see Figure 1 (a)) is one-unambiguous but not strongly one-unambiguous. Consider the bracketing $[_1.\ ([_2.([_3.a.]_3)^*.]_2) \cup ([_4.([_5.b.]_5)^*.]_4)\ .]_1$. The empty word can be matched in two alternative ways, namely as $u.\alpha = [_1.[_2.]_2.]_1$ and as $u.\beta = [_1.[_4.]_4.]_1$. The equivalent regular expression $a.a^* \cup b^*$ is strongly one-unambiguous. □

Example 8. The regular expression $(a^*)^*$ with bracketing $[_1.([_2.([_3.a.]_3)^*.]_2)^*.]_1$ is not strongly one-unambiguous. For instance, for the word $a.a$, there are the bracketings $[_1.[_2.[_3.a.]_3.[_3.a.]_3.]_2.]_1$ and $[_1.[_2.[_3.a.]_3.]_2.[_2.[_3.a.]_3.]_2.]_1$ ($u = [_1.[_2.[_3.a$, $\alpha =]_3.[_3$, $\beta =]_3.]_2.[_2.[_3$, $x = a$, $v = w =]_3.]_2.]_1$). □

Definition 3. An *STDLL(1) Grammar* is an extended regular tree grammar where $\tau(s)$ is unambiguous[5] and in which for each regular expression ρ in the right-hand side of a production, $\tau(\rho)$ is strongly one-unambiguous. □

Obviously, all STDLL(1) grammars are also TDLL(1) grammars.

Remark 2. We suspect that most practical DTDs actually use only strongly one-unambiguous regular expressions in productions and are thus STDLL(1) grammars. Strong one-unambiguity is only a short way from one-unambiguity, and

[5] Obviously, the unambiguity of $\tau(s)$ implies its strong one-unambiguity.

many of the most widely used forms of regular expressions are actually strongly one-unambiguous (e.g., regular expressions of the form $(e_1 \cup \cdots \cup e_m)^*$, where $e_1, \ldots, e_m$ are element names). In particular, the syntactic restriction on mixed-content models mentioned in Remark 1 ensures that such regular expressions are guaranteed to be strongly one-unambiguous. □

2.3 Parse Trees

Given a regular expression π over atomic symbols T, we obtain an equivalent (extended) *regular grammar* $G = (V, T, P, \underline{\pi})$ by recursively decomposing π into productions P,

$$\underline{\rho_1.\rho_2} ::= \underline{\rho_1}\ \underline{\rho_2} \qquad \underline{\rho_1 \cup \rho_2} ::= \underline{\rho_1} \mid \underline{\rho_2} \qquad \underline{\rho^*} ::= \underline{\rho}^*$$

for regular expressions ρ, ρ_1, ρ_2. Here, by $\underline{\rho}$, we refer to a symbol of $V \cup T$ rather than a regular expression. The nonterminals V consist precisely of the symbols $\underline{\rho}$ such that ρ is nonatomic. (For the special case that π is atomic, P is empty and the start symbol $\underline{\pi}$ is allowed to be a terminal[6].)

Regular grammars provide us with a natural way of assigning parse trees to words. For simplicity, we will use interior nodes of the forms "$\cup$", ".", and "$*$", rather than nonterminals $\underline{\rho_1 \cup \rho_2}$, $\underline{\rho_1.\rho_2}$, and $\underline{\rho^*}$, as illustrated in Figure 2 (b).

For TDLL(1) grammars, the parse trees are simply the usual document trees associated to XML documents. However, for STDLL(1) grammars, the parse trees incorporate the parse trees of the regular expressions occurring in productions on the input document. (If a regular expression contains a nonterminal nt, we assign it the terminal t as its unique child in the parse tree if the production used to rewrite nt is of the form $nt ::= t(\rho)$, for some ρ.)

We illustrate the two forms of parse trees by an example.

Example 9. The extended regular tree grammar G of Example 1 is an STDLL(1) grammar. Consider the XML document

$$\langle\text{bib}\rangle\ \langle\text{article}\rangle\ \langle\text{title/}\rangle\ \langle\text{author/}\rangle\ \langle\text{author/}\rangle\ \langle\text{author/}\rangle\ \langle\text{/article}\rangle\ \langle\text{/bib}\rangle$$

As G is also a TDLL(1) grammar, the document parses into the tree depicted in Figure 2 (a).

The parse tree for G viewed as an STDLL(1) grammar is shown in Figure 2 (b). Here we assume that the operation "." associates to the right and that the production

$$\mathit{article} ::= \text{article}\ \ \mathit{title}.\mathit{author}.\mathit{author}^*$$

of G is thus equivalent to

$$\mathit{article} ::= \text{article}\ \ \mathit{title}.(\mathit{author}.\mathit{author}^*)\ .$$

□

[6] This is somewhat nonstandard but fits ours needs.

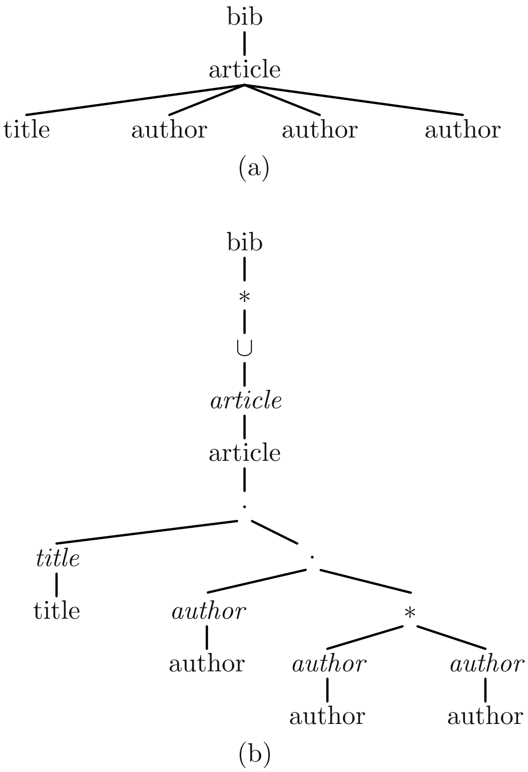


Fig. 2. Parse trees of Example 9.

3 XML Stream Attribute Grammars

We are now in the position to define our main attribute grammar formalism, XML Stream Attribute Grammars (XSAGs).

3.1 XSAGs in the Abstract

Definition 4 (Syntax). Let $Att = \{a_1, \ldots, a_k\}$ be a set of attributes and *Dom* be a finite set of *domain values* (or, to invoke an alternative intuition, of *states*).

Let $F_{\$[}$ denote the class of partial functions

$$f_{\$[} : Dom^k \rightarrow Dom^k \times string,$$

called first-visit attribution functions, and let $F_{\$]}$ denote the partial functions

$$f_{\$]} : Dom^{2k} \rightarrow Dom^k \times string,$$

called second-visit attribution functions. (We will introduce a language for implementing these partial functions in Section 3.2).

A basic XSAG (bXSAG) is an attributed extended regular tree grammar $G = (\mathit{Nt}, T, P, s)$ with nonterminals Nt, terminals $T = \mathit{Tag} \cup \mathit{Char}$, and productions in P that each are of one of the four forms

$$\mathit{nt} ::= t(\rho) \quad \mathit{nt} ::= \{f_{\$[}\}\ t(\rho)$$

$$\mathit{nt} ::= t(\rho)\ \{f_{\$]}\} \quad \mathit{nt} ::= \{f_{\$[}\}\ t(\rho)\ \{f_{\$]}\}$$

where $\mathit{nt} \in \mathit{Nt}$, $t \in T$, $f_{\$[} \in F_{\$[}$, $f_{\$]} \in F_{\$]}$, and ρ is either ϵ or a regular expression over Nt such that $\tau(\rho)$ is one-unambiguous.

The abstract syntax of an *attributed regular expression* over symbols Σ can be specified by the EBNF

$$\begin{aligned} \mathit{aregex} &::= (\text{“\{”}\ F_{\$[}\ \text{“\}”})?\ \ \mathit{aregex}_0\ \ (\text{“\{”}\ F_{\$]}\ \text{“\}”})? \\ \mathit{aregex}_0 &::= \Sigma \mid \mathit{aregex}\ \text{“.”}\ \mathit{aregex} \mid \\ &\qquad \mathit{aregex}\ \text{“}\cup\text{”}\ \mathit{aregex} \mid \mathit{aregex}\ \text{“*”} \end{aligned}$$

An easy XSAG (yXSAG) is an attributed extended regular tree grammar $G = (\mathit{Nt}, T, P, s)$ where each production in P is of one of the four forms

$$\mathit{nt} ::= t(\alpha) \quad \mathit{nt} ::= \{f_{\$[}\}\ t(\alpha)$$

$$\mathit{nt} ::= t(\alpha)\ \{f_{\$]}\} \quad \mathit{nt} ::= \{f_{\$[}\}\ t(\alpha)\ \{f_{\$]}\}$$

where α is either ϵ or an attributed regular expression over symbols Nt such that for the regular expression ρ obtained from α by removing the attributions (enclosed in curly braces), $\tau(\rho)$ is *strongly* one-unambiguous. □

The main purpose of the grammar component of an XSAG is to unambiguously map XML documents to parse trees[7]. The only differences between bXSAGs and yXSAGs are that the former use TDLL(1) grammars while the latter use STDLL(1) grammars, and that in yXSAGs, right-hand side regular expressions may be attributed[8].

It remains to specify how our attribute grammars are evaluated on such parse trees. Obviously, there is a natural method of assigning the attribution functions from $F_{\$[}$ and $F_{\$]}$ to nodes of the parse tree. For the sake of simplicity, we assume that each node v of the parse tree is assigned two functions $f^v_{\$[} \in F_{\$[}$ and $f^v_{\$]} \in F_{\$]}$ through the attribute grammar definition. Where this has not been the case, the defaults are

$$f^d_{\$[} : \langle x_1, \ldots, x_k \rangle \mapsto \langle x_1, \ldots, x_k, \epsilon \rangle$$

and

$$f^d_{\$]} : \langle x_1, \ldots, x_k, x_{k+1}, \ldots, x_{2k} \rangle \mapsto \langle x_{k+1}, \ldots, x_{2k}, \epsilon \rangle.$$

[7] However, for evaluating XSAGs it will not at any time be necessary to maintain entire parse trees in memory.

[8] And indeed, precisely the restriction to STDLL(1) grammars makes it safe to attribute regular expressions in XSAGs.

bXSAGs and yXSAGs are (attributed) *extended* regular tree grammars. For such grammars, nodes of the parse tree may have an arbitrary number of children. When dealing with streams, we generally cannot store the attribute values of all these children in memory. We thus have to introduce special restrictions to be able to deal with streams on the one hand and at the same time assure ease of use and expressiveness to cover practical queries on the other.

We define XSAGs as L-attributed grammars, i.e., attribute grammars whose attributes are evaluated by a single depth-first left-to-right traversal of the document tree. Each node v of the parse tree is visited twice (the visits are referred to by $\$[$ and $\$]$), first from the previous sibling or the parent of v (if v has no previous sibling) and a second time on returning from the rightmost child of v. To provide a clear picture of the necessary computations, we distinguish the states of attribute values *before* (using the subscript "in") and *after* (using the subscript "out") the application of an attribution function.

Definition 5 (Semantics). Let $q_\perp \in \mathit{Dom}$ be a special "uninitialized" value. We evaluate an XSAG on a parse tree $\mathbf{T}$ by a depth-first traversal of $\mathbf{T}$ in which we compute, for each attribute $a_i \in \mathit{Att}$ and each node v of $\mathbf{T}$, the four assignments $(a_i)^v_{\$[.in}$, $(a_i)^v_{\$[.out}$, $(a_i)^v_{\$].in}$, and $(a_i)^v_{\$].out}$ (inductively) as follows.

$$(a_i)^v_{\$[.in} := \begin{cases} q_\perp & \ldots\ v \text{ is the root node} \\ (a_i)^{v_0}_{\$[.out} & \ldots\ v \text{ is the first child of } v_0 \\ (a_i)^{v_0}_{\$].out} & \ldots\ v \text{ is the right sibling of } v_0 \end{cases}$$

$$(a_i)^v_{\$].in} := \begin{array}{ll} (a_i)^v_{\$[.out} & \ldots\ v \text{ has no children} \\ (a_i)^w_{\$].out} & \ldots\ w \text{ is the rightmost child of } v \end{array}$$

In the first visit of node v, we compute

$$\langle (a_1)^v_{\$[.out}, \ldots, (a_k)^v_{\$[.out}, \sigma\rangle := f^v_{\$[}\ \ (a_1)^v_{\$[.in}, \ldots, (a_k)^v_{\$[.in}$$

and write σ to the output. In the second, we compute

$$\langle (a_1)^v_{\$].out}, \ldots, (a_k)^v_{\$].out}, \sigma\rangle := f^v_{\$]}\ \ (a_1)^v_{\$[.in}, \ldots, (a_k)^v_{\$[.in}, (a_1)^v_{\$].in}, \ldots, (a_k)^v_{\$].in}$$

and write σ to the output. In case $f^v_{\$[}$ or $f^v_{\$]}$ is undefined on its input, the evaluation terminates and the input is rejected.

The result of an XSAG on an input tree is the *output* (rather than attribute values) it computes, if it accepts its input. □

Even though this semantics definition may seem involved, we believe that its application is natural.

Example 10. Consider the bXSAG G with $\mathit{Dom} = \{q_\perp, q_{book}, q_{article}\}$, $\mathit{Att} = \{\mathit{prev}\}$, productions

$$\begin{aligned} \mathit{bib} &::= \{f^{bib}_{\$[}\}\ \mathrm{bib}((\mathit{book} \cup \mathit{article})^*)\ \{f^{bib}_{\$]}\} \\ \mathit{book} &::= \{f^{book}_{\$[}\}\ \mathrm{book}(\epsilon) \\ \mathit{article} &::= \{f^{article}_{\$[}\}\ \mathrm{article}(\epsilon) \end{aligned}$$

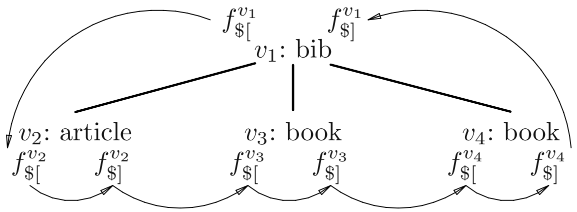


Fig. 3. bXSAG parse tree and traversal of Example 10.

and the attribution functions

$$\begin{aligned} f^{bib}_{\$[} &: x \mapsto (x, \langle\text{bib}\rangle) \\ f^{article}_{\$[} &: x \mapsto (q_{article}, \langle\text{article/}\rangle) \\ f^{book}_{\$[} &: (x_0, x) \mapsto \begin{cases} (q_{book}, \langle\text{book/}\rangle) & \ldots\ x = q_{article} \\ (q_{book}, \epsilon) & \ldots\ \text{otherwise} \end{cases} \\ f^{bib}_{\$]} &: (x_0, x) \mapsto (x, \langle\text{/bib}\rangle) \end{aligned}$$

The grammar requires the input to consist of a dummy bibliography database containing book and article nodes without children. As output, the XSAG writes a root node labeled "bib", to which it assigns bachelor nodes labeled "book" and "article" as children, filtering out books which are not right neighbors of articles[9].

The parse tree of XML document

$$\langle\text{bib}\rangle\ \langle\text{article/}\rangle\ \langle\text{book/}\rangle\ \langle\text{book/}\rangle\ \langle\text{/bib}\rangle$$

is shown in Figure 3. Naturally, we assign $f^{bib}_{\$[}$ to $f^{v_1}_{\$[}$, $f^{bib}_{\$]}$ to $f^{v_1}_{\$]}$, $f^{article}_{\$[}$ to $f^{v_2}_{\$[}$, $f^{book}_{\$[}$ to $f^{v_3}_{\$[}$ and $f^{v_4}_{\$[}$, and have $f^{v_2}_{\$]}, f^{v_3}_{\$]}, f^{v_4}_{\$]} : (x_1, x_2) \mapsto (x_2, \epsilon)$, i.e., the default. (The attribute *prev* is initialized with $q_\perp$.)

G is evaluated on the parse tree as shown in Table 1. The four columns have the following meaning. The first column shows which attribution function is applied in the respective step. The second and third columns show the value of our attribute before and after the application of the attribution function, respectively, and the rightmost column shows which output is produced and written to the output stream. Clearly, G outputs

$$\langle\text{bib}\rangle\ \langle\text{article/}\rangle\ \langle\text{book/}\rangle\ \langle\text{/bib}\rangle$$

and accepts its input. □

[9] This is a somewhat contrived example but illustrates a number of important points related to the evaluation of XSAGs.

Table 1. Run of bXSAG G of Example 10.

	Attribute value		
F	input	output	Output
$f_{\$[}^{v_1}$	$q_\perp$	$q_\perp$	⟨bib⟩
$f_{\$[}^{v_2}$	$q_\perp$	$q_{article}$	⟨article/⟩
$f_{\$]}^{v_2}$	$q_{article}$	$q_{article}$	ϵ
$f_{\$[}^{v_3}$	$q_{article}$	q_{book}	⟨book/⟩
$f_{\$]}^{v_3}$	q_{book}	q_{book}	ϵ
$f_{\$[}^{v_4}$	q_{book}	q_{book}	ϵ (!!)
$f_{\$]}^{v_4}$	q_{book}	q_{book}	ϵ
$f_{\$]}^{v_1}$	q_{book}	q_{book}	⟨/bib⟩

Example 11. Consider again the XSAG of the previous example. Alternatively, to reject the input[10] if two books arrive in sequence, we define $f_{\$[}^{book}$ as

$$f_{\$[}^{book} : x \mapsto \begin{cases} (q_{book}, \langle\text{book/}\rangle) \ldots\ x = q_{article} \\ \text{undefined} \qquad\quad \ldots\ \text{otherwise.} \end{cases}$$

This new XSAG rejects the input document of Example 10. □

3.2 Concrete XSAGs

We introduce a simple imperative programming language for the definition of attribution functions. This language is basically a fragment of Pascal, comprising the following constructs: (1) if-then-else statements, (2) blocks of multiple commands starting with the keyword “begin” and ending with “end”, (3) boolean formulas – using “and”, “or”, and “not” – over equality conditions (used in if-statements), (4) assignments, (5) the keyword “reject” for terminating the computation and rejecting the input, and (6) “print” statements taking a constant string as argument.

An assignment is a statement of the form $x := y$, where x is an l-value and y is an r-value[11]. An equality condition is a statement of the form $x = y$, where x and y are r-values.

The semantics of such a program is defined using the usual notion of an environment, a function $\mathcal{E} : \mathit{Att} \to \mathit{Dom}$ that maps each attribute name to a domain value. For a program defining a function $f_{\$[} : \mathit{Dom}^k \to \mathit{Dom}^k \times \mathit{string}$, at the start of the execution of $f_{\$[}(\boldsymbol{x})$ (where $\boldsymbol{x} = \langle x_1, \ldots, x_k\rangle$), $\mathcal{E}(a_i) = x_i$ for $1 \leq i \leq k$, $\mathit{Att} = \{a_1, \ldots, a_k\}$, where the attributes a_i may be read as well as written (by assignment “:=”). $f_{\$[}(\boldsymbol{x})$ evaluates to $\langle \mathcal{E}^\omega(a_1), \ldots \mathcal{E}^\omega(a_k), o\rangle$, where

[10] This could be alternatively achieved by modifying the grammar rather than the attributions as done in this example, but the goal here is to illustrate the use of partially undefined attribution functions.

[11] We call constructs of our language that may appear on the right side of an assignment *r-values* and and those that may appear on the left side of an assignment *l-values*.

$\mathcal{E}^\omega$ is the environment at the end of the execution and o is the concatenation of the symbols printed. Thus, for (partial) functions in $F_{\$[}$, the l-values consist of the set $\{\$[.a \mid a \in \mathit{Att}\}$ and the r-values consist of $\{\$[.a \mid a \in \mathit{Att}\} \cup \mathit{Dom}$.

For a program defining a function $f_{\$]} : \mathit{Dom}^{2k} \to \mathit{Dom}^k \times \mathit{string}$, at the start of the execution of $f_{\$]}(\$[.\boldsymbol{x}, \$].\boldsymbol{x})$, $\langle \$[.\boldsymbol{x}, \$].\boldsymbol{x} \rangle$ is copied into the environment, but the attributes of $\$[.\boldsymbol{x}$ are read-only (i.e., must not be used on the left-hand sides of assignments). For (partial) functions in $F_{\$]}$, the l-values are $\{\$].a \mid a \in \mathit{Att}\}$ and the r-values are $\{\$[.a, \$].a \mid a \in \mathit{Att}\} \cup \mathit{Dom}$.

Such a program defines functions in $F_{\$[}$ resp. $F_{\$]}$ in the obvious way, with the notable fact that the functions are assumed undefined for inputs for which the "reject" statement is called.

Example 12. Using our Pascal-like syntax, we define the attribution functions of Example 10 as

$$\begin{aligned}
f_{\$[}^{bib} &= \{\text{print } \langle \text{bib} \rangle\} \\
f_{\$]}^{bib} &= \{\text{print } \langle /\text{bib} \rangle\} \\
f_{\$[}^{article} &= \{\text{print } \langle \text{article}/ \rangle; \$[.\mathit{prev} := q_{article}\} \\
f_{\$[}^{book} &= \{\text{if } \$[.\mathit{prev} = q_{article} \text{ then print } \langle \text{book}/ \rangle; \\
&\qquad \$[.\mathit{prev} := q_{book}\}
\end{aligned}$$

Thus, we can write the bXSAG of Example 10 as

$$\begin{aligned}
\mathit{bib} &::= \{\text{print } \langle \text{bib} \rangle\} \text{ bib}((\mathit{book} \cup \mathit{article})^*) \; \{\text{print } \langle /\text{bib} \rangle\} \\
\mathit{book} &::= \{\text{if } \$[.\mathit{prev} = q_{article} \text{ then print } \langle \text{book}/ \rangle; \\
&\qquad \$[.\mathit{prev} := q_{book}\} \text{ book}(\epsilon) \\
\mathit{article} &::= \{\text{print } \langle \text{article}/ \rangle; \$[.\mathit{prev} := q_{article}\} \text{ article}(\epsilon)
\end{aligned}$$

To modify the XSAG to reject its input if two books arrive in sequence on the stream as in Example 11, we define $f_{\$[}^{book}$ as

$$\{\text{if } \$[.\mathit{prev} = q_{article} \text{ then begin print } \langle \text{book}/ \rangle; \; \$[.\mathit{prev} := q_{book} \text{ end else reject}\}$$

□

3.3 Built-in Macros

For the convenient definition of queries using XSAGs, we introduce three standard built-in macros, *ECHO*, *ECHO_OFF*, and *MATCH_CHILDREN*. These are redundant with the formalism presented so far, but allow to define queries in a more concise way.

If macro *ECHO* is used in a first-visit attribution function, the subtree of the parse tree to which the attribution function applies is copied to the output. Correspondingly, macro *ECHO_OFF* can be used to override *ECHO* and to suppress the output of certain XML subtrees. Example 1 illustrates the use of the macro *ECHO*. Below, we show an example that combines *ECHO* and *ECHO_OFF*.

Example 13. The bXSAG

$$\begin{aligned} \mathit{bib} &::= \{\mathit{ECHO}\}\ \text{bib}\ \ \mathit{book}^* \\ \mathit{book} &::= \text{book}\ \ \mathit{title}.\mathit{author}.(\{\mathit{ECHO\_OFF}\}\ \mathit{author}^*) \\ \mathit{title} &::= \text{title}(\mathit{PCDATA}) \\ \mathit{author} &::= \text{author}(\mathit{PCDATA}) \end{aligned}$$

outputs each book with its title and first author (dropping further authors). □

To realize *ECHO*, we define a boolean attribute $\mathit{echo} \in \mathit{Att}$, initialized with *false*. Occurrences of *ECHO* are replaced by $\$[.\mathit{echo} := \mathit{true}$ and occurrences of *ECHO_OFF* are replaced by $\$[.\mathit{echo} := \mathit{false}$.

For every production $\{f_{\$[}\}\ t(\rho)\ \{f_{\$]}\}$,

- if $t \in \mathit{Tag}$, we append

  $$\text{if } \$[.\mathit{echo} = \mathit{true} \text{ then print } \langle \text{t} \rangle$$

  to $f_{\$[}$ and

  $$\text{if } \$[.\mathit{echo} = \mathit{true} \text{ then print } \langle /\text{t} \rangle$$

  to $f_{\$]}$.
- If $t \in \mathit{Char}$ we append "if $\$[.\mathit{echo} = \mathit{true}$ then print t" to $f_{\$[}$.

In both cases, we finally append "$\$].\mathit{echo} := \$[.\mathit{echo}$" to $f_{\$]}$. On leaving a node v for which $f^v_{\$[}$ contains *ECHO* or *ECHO_OFF*, attribute *echo* is reset to its former value. Thus, even though *ECHO* may be overridden by *ECHO_OFF* within a subtree, we cannot accidentally create malformed documents.

The *MATCH_CHILDREN*(ρ, c) macro matches the string obtained by concatenating the character data encountered when traversing the children of a node from left to right against a regular expression ρ, yielding *true* or *false* as a value for the user-defined condition attribute c. *MATCH_CHILDREN* may only be used in first-visit attribution functions. The evaluation result becomes available in the corresponding second-visit attribution function.

Example 14. We slightly extend Example 3. The yXSAG production

$$\begin{array}{ll} \mathit{book} ::= \text{book} & (\{\mathit{MATCH\_CHILDREN}(2003, \$[.c)\}\mathit{year}). \\ & (\{\text{if } \$[.c = \mathit{true} \text{ then} \\ & \quad \text{begin} \\ & \qquad \text{print } \langle\text{book}\rangle;\ \mathit{ECHO} \\ & \quad \text{end}\} \\ & \ (\mathit{title}.\mathit{author}.\mathit{author}^*) \\ & \ \{\text{if } \$].c = \mathit{true} \text{ then print } \langle\text{year}\rangle 2003 \langle/\text{year}\rangle\langle/\text{book}\rangle\}) \end{array}$$

selects those books for which child *year* has string value "2003"; moreover, the year is output as the rightmost child of book, rather than as the leftmost as required for the input. (We omit productions defining *bib*, *year*, *title*, and *author*, which are as in Example 15.) □

MATCH_CHILDREN can be easily implemented by compiling ρ into a DFA and running it on the characters encountered while traversing a node's children.

3.4 bXSAGs vs. yXSAGs

As we show in the next section, bXSAGs and yXSAGs have the same expressive power. However, yXSAGs are more convenient to use. In particular, it is often necessary to introduce more attributes and more complicated attribution functions to encode a given query as a bXSAG than to encode it as a yXSAG.

Example 15. Consider the yXSAG with productions

$$\begin{aligned}
\mathit{bib} &::= \{\text{print } \langle\text{bib}\rangle\}\ \text{bib}(\mathit{article}^{*})\ \{\text{print } \langle/\text{bib}\rangle\}\\
\mathit{article} &::= \text{article}\quad \{\text{print } \langle\text{article\_short}\rangle; \mathit{ECHO}\}\\
&\qquad\qquad\qquad (\mathit{title}.\mathit{author}.\mathit{author}^{*})\\
&\qquad\qquad\quad \{\text{print } \langle/\text{article\_short}\rangle\}\\
&\qquad\qquad\quad \cup\\
&\qquad\qquad\quad \{\text{print } \langle\text{article\_long}\rangle; \mathit{ECHO}\}\\
&\qquad\qquad\qquad (\mathit{year}.\mathit{title}.\mathit{author}.\mathit{author}^{*}.\mathit{pub})\\
&\qquad\qquad\quad \{\text{print } \langle/\text{article\_long}\rangle\}\\
\mathit{title} &::= \text{title}(\mathit{PCDATA})\\
\mathit{author} &::= \text{author}(\mathit{PCDATA})\\
\mathit{year} &::= \text{year}(\mathit{PCDATA})\\
\mathit{pub} &::= \text{pub}(\mathit{PCDATA})
\end{aligned}$$

Article entries can appear either in a short version with title and authors only, or in a long version which also contains a year and a publisher. In the former case, *article* nodes are relabeled as *article_short*, and in the latter case, *article* nodes are relabeled *article_long*.

The following bXSAG is equivalent to the above yXSAG:

$$\begin{aligned}
\mathit{bib} &::= \{\text{print } \langle\text{bib}\rangle\}\ \text{bib}(\mathit{article}^{*})\ \{\text{print } \langle/\text{bib}\rangle\}\\
\mathit{article} &::= \{\$[.\mathit{state} := q_{\mathit{unknown}}\}\\
&\qquad \text{article}\ \ (\mathit{title}.\mathit{author}.\mathit{author}^{*}) \cup\\
&\qquad\qquad\quad (\mathit{year}.\mathit{title}.\mathit{author}.\mathit{author}^{*}.\mathit{pub})\\
&\qquad \{\text{if } \$].\mathit{state} = q_{\mathit{short}} \text{ then print } \langle/\text{article\_short}\rangle\\
&\qquad\ \text{else if } \$].\mathit{state} = q_{\mathit{long}} \text{ then print } \langle/\text{article\_long}\rangle\}\\
\mathit{title} &::= \{\text{if } \$[.\mathit{state} = q_{\mathit{unknown}} \text{ then}\\
&\qquad\ \text{begin } \$[.\mathit{state} := q_{\mathit{short}};\ \text{print } \langle\text{article\_short}\rangle \text{ end}; \mathit{ECHO}\}\\
&\qquad \text{title}(\mathit{PCDATA})\\
\mathit{author} &::= \{\mathit{ECHO}\}\ \text{author}(\mathit{PCDATA})\\
\mathit{year} &::= \{\$[.\mathit{state} := q_{\mathit{long}};\ \text{print } \langle\text{article\_long}\rangle;\ \mathit{ECHO}\}\\
&\qquad \text{year}(\mathit{PCDATA})\\
\mathit{pub} &::= \{\mathit{ECHO}\}\ \text{pub}(\mathit{PCDATA})
\end{aligned}$$

While there are other ways of encoding our query as a bXSAG, it does not seem to be possible to represent the query as a bXSAG without using attributes. □

It is easy to verify that bXSAGs equivalent to the yXSAGs of Examples 2 and 14 are also much more complicated.

4 Expressive Power of XSAGs

We introduce deterministic pushdown transducers (*DPDT*s) as deterministic pushdown automata with output which accept by empty stack. As stated in [2], the *DPDT*s accepting by empty stack are equivalent to the *DPDT*s accepting by final state.

Definition 6 (*DPDT*). A *deterministic pushdown transducer* is a tuple

$$\mathcal{T} = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0)$$

where Q is a finite set of states, Σ, Γ, and Δ are the finite alphabets for input tape, stack, and output tape respectively, δ is the partial transition function

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^* \times \Delta^*,$$

$q_0 \in Q$ denotes the initial state, and Z_0 the initial stack symbol. For each $q \in Q$ and $X \in \Gamma$ s.t. $\delta(q, \epsilon, X)$ is defined, $\delta(q, a, X)$ is undefined for all $a \in \Sigma$.

We define a run of $\mathcal{T}$ by means of *instantaneous descriptions* (IDs). An ID is a quadruple

$$(q, w, \alpha, o) \in Q \times \Sigma^* \times \Gamma^* \times \Delta^*,$$

where q is a state, w is the remaining input, α a string of stack symbols, and o the output generated so far. We make a transition

$$(q, aw, X\alpha, o) \vdash (q', w, \gamma\alpha, o\sigma)$$

if $\delta(q, a, X) = (q', \gamma, \sigma)$ where $a \in \Sigma \cup \{\epsilon\}$, $X \in \Gamma$, $q' \in Q$, and $\sigma \in \Delta^*$.

Here, $\gamma \in \Gamma^*$ is the string of stack symbols which replace X on top of the stack. For $\gamma = \epsilon$, the stack is popped, whereas for $\gamma = X$, the stack remains unchanged. If $\gamma = YX$, then Y is pushed on top of X.

Let $\vdash^*$ be the reflexive and transitive closure of $\vdash$. $\mathcal{T}$ accepts by empty stack if

$$(q_0, w, Z_0, \epsilon) \vdash^* (q, \epsilon, \epsilon, o)$$

for $w \in \Sigma^*, q \in Q$, and $o \in \Delta^*$. We say o is the output for input word w.

The translation defined by an *XML-DPDT* $\mathcal{T}$, denoted $T(\mathcal{T})$, is

$$\{(w, o) \mid (q_0, w, Z_0, \epsilon) \vdash^* (q, \epsilon, \epsilon, o) \textit{ for some } q \in Q\},$$

We call two DPDTs equivalent if they define the same translation. □

Definition 7 (Well-formedness). An XML document is called *well-formed* iff it conforms to an extended regular tree grammar $G = (Nt, T, P, s)$ where for all productions of the form $s ::= t(\rho)$, $t \in \mathit{Tag}$. □

A well-formed document contains at least one element (i.e. the root element) and has element start-tags and end-tags properly nested within each other. Furthermore, the first symbol in the document must be the opening tag for the root node. An XML document is *malformed* if it is not well-formed.

Definition 8 (*XML-DPDT*). Let input alphabet $\Sigma = \{\langle t\rangle, \langle /t\rangle \mid t \in \mathit{Tag}\} \cup \mathit{Char}$ consist of matching opening and closing tags and characters.

Throughout this paper, for a set S, we use $S^{\leq 2}$ as a shortcut for $\{\epsilon\} \cup S \cup S \times S$.

An *XML-DPDT* is a *DPDT*

$$\mathcal{T} = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0)$$

which rejects malformed XML documents and for which the transition function δ is restricted as follows:

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^{\leq 2} \times \Delta^*$$

- In the very first transition the initial stack symbol Z_0 is replaced; i.e., we require $\delta(q_0, \langle t\rangle, Z_0) = (p, Y, \sigma)$ for $\langle t\rangle \in \Sigma$, $p \in Q$, $Y \in \Gamma$, and $\sigma \in \Delta^*$.
- For all other configurations of $q \in Q$ and $X \in \Gamma$, a symbol is only pushed on the stack when an opening tag is read from the input stream: $\delta(q, \langle t\rangle, X) = (p, YX, \sigma)$ for $\langle t\rangle \in \Sigma$, $p \in Q$, $Y \in \Gamma$, and $\sigma \in \Delta^*$.
- A symbol is only popped from the stack when a closing tag is encountered in the input stream, i.e. $\delta(q, \langle /t\rangle, X) = (p, \epsilon, \sigma)$ for $q, p \in Q$, $\langle /t\rangle \in \Sigma$, $X \in \Gamma$, and $\sigma \in \Delta^*$. □

The conditions required in the definition of *XML-DPDT* are only natural in the context of XML stream processing: The size of the stack is bounded by the maximum depth of the incoming document tree. Moreover, the input has to start with the root element of the XML document being read. Due to acceptance by empty stack, only well-formed XML documents are accepted.

Let $L(G)$ be the language accepted by XSAG G. For *input* $w \in L(G)$, $G(w)$ is the *output* produced by G in accepting w. The *translation* defined by an XSAG G, written $T(G)$, is $\{(w, o) \mid w \in L(G) \textit{ and } o \in G(w)\}$. We call XSAG G and *XML-DPDT* $\mathcal{T}$ equivalent if they define the same translation, i.e. $T(G) = T(\mathcal{T})$.

Both bXSAGs and yXSAGs are precisely as expressive as *XML-DPDT*s [12].

Theorem 1. *For each bXSAG, there is an equivalent XML-DPDT.*

Theorem 2. *For each XML-DPDT, there is an equivalent XSAG which is both a bXSAG and a yXSAG.*

Theorem 3. *For each yXSAG, there is an equivalent XML-DPDT.*

The lengthy proofs for these three theorems will be provided in the long version of this paper.

[12] Note, however, that there are bXSAGs that are no yXSAGs and vice versa, so Theorems 1 and 3 are not redundant.

5 Efficient Evaluation of XSAGs

The proofs of Theorems 1 and 3 are based on a translation to DPDTs which, as a method for evaluating XSAGs, has the strong point that once the DPDT has been created, the query evaluation time is in principle independent of the size of the XSAG/DPDT and only depends on the input data.

Corollary 1. *An XSAG G can be evaluated on a tree T in time $O(f(|G|)+|T|)$ using only a stack of memory of size $O(depth(T))$.*

However, the DPDTs of the construction of the proof of Theorem 1 are of size *exponential* in the number k of attributes in the XSAG, i.e., f is $O(2^k)$.

The exponential-time compilation phase can be avoided by using a simple hybrid evaluation method in which the grammars (and in particular the regular expressions appearing in the grammar productions) are compiled into transducers which however *interpret* the attribution functions (rather than materializing the *graphs* of the attribution functions as is done in our proofs). Thus one obtains an XSAG evaluation method which runs scalably on streams and which is strictly polynomial in the size of the XSAG.

Theorem 4. *A bXSAG G can be evaluated on a tree T in time $O(|G|^2+|T|\cdot|G|)$ using a stack of size $O(depth(T))$.*

Theorem 4 also makes use of the fact that DFAs for one-unambiguous regular expressions can be computed in polynomial (actually, quadratic) time[13] [6].

For yXSAGs, the construction of the proof of Theorem 3 is in addition exponential in the maximum depth of the parse trees of the regular expressions used (which only depend on the XSAG). This can be resolved by pushing attributes onto the stack at yXSAG regular expression nodes as well. The stack consumption of course remains proportional to the depth of the input tree.

The main technical challenge we have to deal with when evaluating yXSAGs is the matching of attributed regular expressions on the stream and the invocation of attribution functions at the right time.

A finite-state transducer (FST) is an NFA with output, which in each transition $q \overset{a/w}{\to} q'$ from state q to q' on input symbol a outputs a fixed word w. A deterministic finite-state transducer (DFT) is an FST that is deterministic, i.e., which is a DFA if the output is ignored and for which no two transitions $q \overset{a/v}{\to} q'$ and $q \overset{a/w}{\to} q'$ exist such that $v \neq w$.

Below, in regular expressions of the form $\rho.\odot$, let $\odot$ be a new end-marker symbol that does not occur in ρ.

[13] This construction – that of the Glushkov automaton of a regular expression – also provides a procedure for deciding one-ambiguity with the same complexity. A regular expression is known to be one-unambiguous precisely if its Glushkov automaton is deterministic.

Theorem 5. *Let ρ be a regular expression. Then, there is an FST $\mathcal{A}^{[]}(\rho)$ which*

1. *recognizes $L(\rho.\odot)$,*
2. *is deterministic iff ρ is strongly one-unambiguous,*
3. *if ρ is strongly one-unambiguous, outputs the bracketing of word w for each $w.\odot \in L(\rho.\odot)$, and*
4. *can be computed in time $O(|\rho|^3)$.*

Using the DFT construction of Theorem 5, the preprocessing phase for yXSAGs takes time cubic in the size of each of the productions.

Theorem 6. *A yXSAG G can be evaluated on a tree T in time $O(|G|^3 + |T| \cdot |G|)$ using a stack of size $O(depth(T) \cdot |G|)$.*

6 Discussion and Conclusions

The goal of this paper was to develop a framework for query formulation which

1. satisfies our criteria for scalable query processing on streams,
2. has a good and well-justified foundation, and
3. is user-friendly, i.e. allows to state many common queries quickly and easily.

We can argue that XSAGs satisfy these three desiderata.

(1) Each XSAG can be translated into a DPDT with a stack discipline that assures that the size of the stack remains proportional to the depth of the XML tree. This is known to be the minimum amount of memory required to do any meaningful (sequential) processing of XML data [10]. Of course, queries are evaluated strictly in linear time.

(2) Throughout the paper, we have explained and justified our design choices. Regular tree grammars are a commonly accepted grammar formalism for XML documents (as are DTDs, which are restricted regular tree grammars). In the right-hand sides of the productions of such grammars, we use regular expressions to be able to parse nodes with an unbounded number of children. Our restriction of these regular expressions to strongly one-unambiguous ones in the case of yXSAGs allows for precisely those expressions for which the parse trees of words can be unambiguously generated using a lookahead of only one symbol (a necessity in stream processing). Having the regular expressions inside grammar productions available for attribution allows to conveniently define attribute grammars for unranked trees, and to approach the usability of XML Query with a formalism that allows for much better control of complexity.

We have precisely characterized the expressive power of XSAGs in relation to deterministic pushdown transducers.

Note that our formalism fully fits into the classical framework of attribute grammars (and more precisely, L-attributed grammars), even if we did not introduce, say, the distinction between synthesized and inherited attributes.

(3) A number of examples in this paper and our experiences with many more demonstrate that XSAGs are of practical value, and that they fill an important void in the design space of tailored query languages.

Earlier in this paper, we defined XSAGs with attributes ranging exclusively over a finite domain to be able to assure scalability and memory bounds in the strongest sense. However, it is desirable and often justified to generalize this framework to make certain uniformity assumptions and to allow for values from an infinite domain. In the future, we plan to carry out a more detailed study of conservative extensions of our formalism with small buffers (using uniformity assumptions for numbers, small strings, and small subtrees).

Acknowledgments

We thank Peter Buneman, Nicole Schweikardt, Bernhard Stegmaier, Val Tannen, and the anonymous reviewers of DBPL 2003 for their suggestions, which helped to substantially improve the presentation of the paper.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling. I: Parsing*, volume 1. Prentice-Hall, 1972.
3. M. Benedikt, C. Y. Chan, W. Fan, J. Freire, and R. Rastogi. "Capturing both Types and Constraints in Data Integration". In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pages 277–288, 2003.
4. M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. "DTD-Directed Publishing with Attribute Translation Grammars". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 838–849, 2002.
5. T. Bray, J. Paoli, and C. Sperberg-McQueen. "Extensible Markup Language (XML) 1.0". Technical report, W3C, Feb. 1998.
6. A. Brüggemann-Klein and D. Wood. "One-Unambiguous Regular Languages". *Information and Computation*, **142**(2):182–206, 1998.
7. L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. "Query Processing of Streamed XML Data". In *Proc. 11th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 126–133, 2002.
8. T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. "Processing XML Streams with Deterministic Automata". In *Proc. of the 9th International Conference on Database Theory (ICDT'03)*, 2003.
9. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Reading, MA USA, 1979.
10. C. Koch. "Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach". In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, Berlin, Germany, 2003.
11. D. Lee, M. Mani, and M. Murata. "Reasoning about XML Schema Languages using Formal Language Theory". Technical Report RJ 10197 Log 95071, IBM Research, Nov. 2000.

12. B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. "A Transducer-Based XML Query Processor". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 227–238, 2002.
13. F. Neven. "Extensions of Attribute Grammars for Structured Document Queries". In *Proc. 8th International Workshop on Database Programming Languages (DBPL)*, pages 99–116, 1999.
14. F. Neven and J. van den Bussche. "Expressiveness of Structured Document Query Languages Based on Attribute Grammars". *Journal of the ACM*, **49**(1):56–100, Jan. 2002.
15. D. Olteanu, T. Kiesling, and F. Bry. "An Evaluation of Regular Path Expressions with Qualifiers against XML Streams". In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE)*, Bangalore, Mar. 2003. Poster Session.
16. World Wide Web Consortium. "XML Query". `http://www.w3c.org/XML/query/`.

A General Framework for Estimating XML Query Cardinality

Carlo Sartiani

Dipartimento di Informatica - Università di Pisa
Via Buonarroti 2, Pisa, Italy
Phone: +39 05022133140 – Fax: +39 0502212726
sartiani@di.unipi.it

Abstract. In the context of XML data management systems, the estimation of query cardinality is becoming more and more important: the information provided by a query result estimator can be used as input to the query optimizer, as an early feedback to user queries, as well as input for determining an optimal storage schema, and it may be helpful in embedded query execution.

Existing estimation models for XML queries focus on particular aspects of XML querying, such as the estimation of path and twig expression cardinality, and they do not deal with the problem of predicting the cardinality of general XQuery queries. This paper presents a framework for estimating XML query cardinality. The framework provides facilities for estimating result size of FLWR queries, hence allowing the model designer to concentrate her efforts on the development of adequate and accurate, while concise, statistic summaries for XML data. The framework can also be used for extending existing models to a wider class of XML queries.

1 Introduction

The last few years have seen the rapid emerging of the eXtensible Markup Language (XML). Given its ability to represent nearly any kind of information, XML imposes itself as the standard format for representing *semistructured* data, and it is also increasingly used as data exchange format.

In the context of XML data management systems, the estimation of query cardinality is becoming more and more important: the information provided by a query result estimator can be used as input to the query optimizer, as an early feedback to user queries, as well as input for determining an optimal storage schema [5], and it may be helpful in embedded query execution.

As for many other common database tasks, the peculiar nature of XML makes result size estimation more difficult than in the relational context: in particular, the (very) non-uniform distribution of tags and data, together with the dependencies imposed by the tree structure of XML data, make not so reasonable the usual hypothesis used in relational size estimators.

G. Lausen and D. Suciu (Eds.): DBPL 2003, LNCS 2921, pp. 257–277, 2004.

1.1 Our Contribution

This paper describes a framework for estimating the cardinality of FLWR XQuery queries. Existing estimation models for XML queries focus on particular issues of result size estimation: in particular, some models are limited to path expression estimation only [2], while others also deal with twig queries [3] [8], but still ignoring critical issues such as iterators, binders, nested queries, etc. The proposed framework, instead, offers algorithms for estimating the cardinality of full FLWR queries, with the only exception of universally quantified predicates.

Furthermore, the algorithms in the framework estimate not only the *raw* cardinality of query results, but also their distribution, while existing models, with the only notable exception of the StatiX model [5], return only raw cardinalities (e.g., number of tuples in the result).

The framework allows the model designer to concentrate her efforts on the development of adequate and accurate, while concise, statistic summaries for XML data. The framework forms the basis of the estimation model of Xtasy [6], and it can also be used for extending existing models to a wider class of XML queries.

1.2 Paper Outline

The paper is organized as follows. Section 2 describes the main issues in XML query result size estimation. Section 3, then, introduces the framework and its estimation policy, and provides an informal description of the framework algorithms. Next, Section 4 illustrates some experimental results about the estimation model of Xtasy, which is an instance of the framework being described in this paper. Section 5, then, presents the state of the art in XML query cardinality estimation. In Section 6, finally, we draw our conclusions.

2 Issues in Result Size Estimation

Referring to the FLWR fragment of XQuery, the most problematic aspects in result size prediction concern the estimation of path and twig cardinality, the estimation of predicate selectivity, as well as the estimation of group cardinality (let binder of XQuery). While path and twig estimation is a peculiar issue of XML and semistructured query languages, predicate and group cardinality estimation are well-known problems in database theory and practice. Nevertheless, these problems receive new strength from the irregular nature of XML, as briefly discussed above.

Irregular Tree or Forest Structure. XML data can be seen as node-labeled trees or forests; these trees, being commonly used for representing semistructured data, usually have a deeply nested structure, and are far from being well-balanced. Moreover, the same tag may occur in different parts of the same document with a different semantics, e.g., the tag name under person and the tag name under city.

The irregular and overloaded structure of XML documents influences cardinality estimation, since the location of a node inside a tree may determine its semantics, and, then, its relevance in operations like path and predicate evaluation. For example, consider the XML document shown in Fig. 1.

```
<root>
  <persons>
     <person>
        <name>
           <fullname> Caius Julius Caesar
           </fullname>
           <gensname> Julia </gensname>
        </name>
     </person>
  </persons>
  <cities>
    <city>
       <name>
          <ancientName> Roma </ancientName>
          <modernName> Roma </modernName>
       </name>
       <nick> Caput Mundi </nick>
       <nick> Eternal City </nick>
    </city>
    <city>
       <name>
          <ancientName> Pisae </ancientName>
          <modernName> Pisa </modernName>
       </name>
    </city>
    <city>
       <name> New York </name>
       <nick> The Big Apple </nick>
    </city>
  </cities>
</root>
```

Fig. 1. A sample XML document

The structure of the name element under person is quite different from the semantics of New York's name, hence the evaluation of any query operation starting from name elements should take this into account.

Non-uniform Distribution of Tags and Values. The irregular structure of XML data, together with their hierarchical tree-shaped nature, leads to the *non-uniform* distribution of tags and values in XML trees. XML non-uniformity is further strengthened by the presence of structural dependencies among elements (e.g., name depends on person, etc). As a consequence, a prediction model should track the provenance of estimated matching elements.

These typical features of XML influence the nature and the “complexity” of the previously cited estimation problems, and give rise to new requirements for prediction models. Hence, a closer look to these problems is necessary.

Path and Twig Cardinality Estimation. Path and twig expressions are used in XQuery and in many other XML query languages for retrieving nodes from a XML tree, and for binding them to variables for later use.

The main difficulties in cardinality estimation for path and twig expressions come from the need to reduce the prediction errors induced by joins (paths and twigs are usually translated in sequences of joins), and, for twigs only, from the need to correlate results coming from different branches.

Predicate Selectivity Estimation. The estimation of predicate selectivity is a well-known problem in database theory and practice. The most effective and accurate solutions rely on histograms for capturing the distribution of values in the data, and on the use of the uniform distribution when nothing is known about the data involved in the predicate.

In the context of XML, predicate selectivity estimation poses new challenges. First, XML data are usually distributed in a (very) non-uniform way, hence the use of the uniform distribution can lead to many potential errors. Second, the selectivity of a predicate such as $data(\$n)\ \theta\ value$ depends not only on θ and $value$, but also on a) the nodes bound to $\$n$, which may be heterogeneous, b) the semantics of those nodes (e.g., name under person is quite different from name under city), and c) the “region” of the document where those nodes appear.

Many existing prediction models (including [3], [8], and [2]), while very sophisticated and accurate, return raw numbers as result of the estimation. Raw numbers, denoting the cardinality of matching nodes in the data tree, do not carry sufficient information for the estimation of subsequent predicates being accurate, hence making the enclosing models not so accurate.

Groups. As noted about nested queries, XQuery misses explicit constructs for performing groupby-like operations[1]. Nevertheless, the let binder can be used for creating heterogeneous sets of nodes, hence for building, together with nested queries, groups and partitions. The let binder, unlike the for binder, accumulates each node returned by its argument into a set, which is then bound to the binding variable. For example,

```
for $c in input()//city,
let $n_list := $c/name,
```

returns, for each city, the list of its names (ancient as well as modern ones).

[1] We are aware of proposals, both public and private to the W3C XQuery Working Group, for extending XQuery with explicit group-by constructs. We delay the extension of the framework until they become more stable.

Estimating the cardinality of the let binder requires the system to a) estimate the number of distinct groups created, b) correlate each group to the variables on which it depends (*$n_list* depends on *$c*), and c) estimate the distribution of nodes and values into each group. This information is necessary since the groups created by the let binder can be used as starting point for further navigational operations, as argument for aggregate functions or for predicates.

The estimation of group cardinality is one of the missing points in current XML prediction model. For what is known to the author, no existing model for XML query languages faces this problem, hence the support of group cardinality estimation at the framework level becomes a *must*.

3 The Framework

3.1 Basics

The main idea behind the framework is to estimate the distribution of data into the result of any query subexpression. Hence, an estimation function based on the framework takes as input a query subexpression (e.g., a node of the query AST, or, as in the model of Xtasy, a node of the physical query plan), as well as a data structure, called ETLS, describing the distribution of the input data, and it returns a new ETLS for the result: this ETLS is obtained by recursively traversing the query subexpression, and by using path and predicate statistics for interpreting for, let, and where clauses. ETLS structures will be described in Section 3.3. Inside ETLSs, data distribution is described by means of sequences of *match occurrences*, which are themselves illustrated in Section 3.3.

We stress that the framework is in some way independent from specific statistics: it acts as a *metamodel*, on top of which specific models, as the Xtasy model, can be built; models conforming to the framework can benefit from the services offered by the framework.

3.2 Tagged Region Graph

Statistic models for database queries usually rely on the partitioning of the database into *regions*, which are used for limiting the scope of aggregated statistics, and, then, for increasing their accuracy. In the proposed framework, regions are defined as follows.

Definition 1. *Given a document $\mathcal{T}$, a region partitioning scheme for $\mathcal{T}$ is a pair $(\mathcal{R}, \mathcal{F})$, where $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$, and $\mathcal{F}$ is a function mapping $\mathcal{T}$ nodes into regions $\mathcal{R}_i$ such that, for any node $x \in \mathcal{T}$ $\mathcal{F}(x) = \mathcal{R}_i \in \mathcal{R}$.*

The notion of region is wide enough to accomplish the needs of different prediction models: it may be the type of the node (*intensional* region), the type of the node together with its location in the originating document (*mixed* region, e.g., the node type and its bucket in the corresponding *structural* histogram in StatiX [5]), or the location of the node in the originating database (*extensional* region, e.g., the grid cell in the related *position* histogram in TIMBER [8]). Regions, hence, can express both intensional and extensional concepts, and are the main tool for describing the distribution of data.

Depending on the kind of partitioning used, regions may overlap: for instance, in a purely intensional partitioning based on a type system with subtyping, person regions may contain student regions. When extensional information comes to play, regions are defined as disjoint, in order to ease the construction of proper histograms.

Regions can be further specialized by partitioning them according to the element tags they contain.

Definition 2. *Given a region $\mathcal{R}_i$ of the document $\mathcal{T}$, $\mathcal{R}_i$ can be split into a set of* tagged regions $\bar{\mathcal{R}} = \{(l_1, \mathcal{R}_i), \ldots, (l_p, \mathcal{R}_i)\}$*, such that $l_1, \ldots, l_p$ are the tags of the elements occurring in $\mathcal{R}_i$, and $\mathcal{R}_i = \cup_{l_j}(l_j, \mathcal{R}_i)$.*

Tagged regions carry more information than regions, since element labels are explicitly indicated. The explicit indication of the label may seem unnecessary, since the region itself may identify the main characteristics of the nodes. This is only partially true. If the partitioning scheme used is intensional and based on DTD-like types, where a 1-1 correspondence between tags and types exists, the label is useless; instead, if there is no 1-1 correspondence between tags and types, the label component becomes necessary.

Tagged regions can be organized to form a graph, the Tagged Region Graph, which is the main statistical structure of the framework. Graph edges are determined by the actual region partitioning scheme as well as by the statistics used in the specific model. In the model of Xtasy, for instance, we use parent/child path statistics, e.g., we store the average number of nodes in the tagged region (l_2, r_2) being sub-elements of nodes in (l_1, r_1) $(N_{afo}((l_1, r_1), (l_2, r_2)))$. Hence, we can define the Tagged Region Graph as follows.

Definition 3. *Given a document $\mathcal{T}$ and a region partitioning scheme $(\mathcal{R}, \mathcal{F})$, the* tagged region graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *is a directed graph, where nodes are labeled with tagged regions (l, r), and $((l, r), (l^{'}, r^{'})) \in \mathcal{E} \iff$ there exists a node $y \in (l^{'}, r^{'})$ and a node $x \in (l, r)$ such that y is a child of x in $\mathcal{T}$.*

The following example shows a sample tagged region graph.

Example 1. Consider the sample document of Fig. 1, obeying the schema of Fig. 2. Assume that we build an intensional partitioning scheme based on such schema. The corresponding tagged region graph $\mathcal{G}$, then, is shown in Fig. 3; statistical information related to these graph is shown in Tables 1 an 2[2].

The tagged region graph plays a key role in the framework, since many algorithms work on it, and it is the natural repository for *region-related* statistics; for instance, the Xtasy model associates each tagged region with its cardinality $(N_{el}(l, r))$, as well as with parent/child statistics $(N_{afo}((l, r), (l^{'}, r^{'})))$.

[2] Tags without explicit types are mapped into the *Any* type.

```
type DB = root[persons[Person*],cities[City*]]
type Person = person[PersonName]
type PersonName = name[fullname[String],
                       gensname[String]?]
type City = city[OldCityName,OldCityNick*|
                 NewCityName,NewCityNick*]
type OldCityName = name[ancientName[String]+,
                        modernname[String]]
type OldCityNick = nick[String]
type NewCityNick = nick[String]
type NewCityName = name[String]
```

Fig. 2. A schema for the sample document

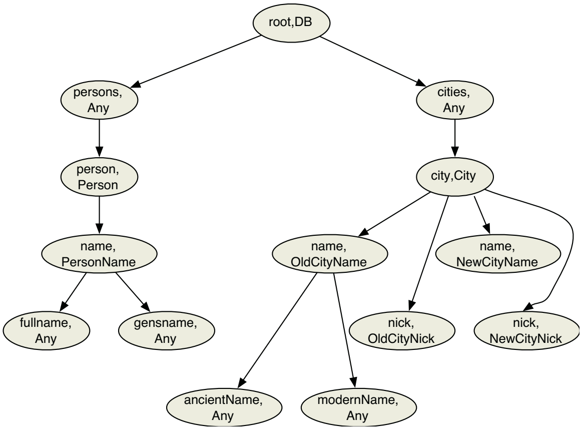


Fig. 3. Tagged region graph with /-edges

Table 1. N_{el} table (Xtasy model) for the tagged region graph of Fig. 3

(root, DB)	1
(persons, Any)	1
(person, Person)	1
(name, PersonName)	1
(fullName, Any)	1
(gensName, Any)	1
(cities, Any)	1
(city, City)	3
(name, OldCityName)	2
(ancientName, Any)	2
(modernName, Any)	2
(nick, OldCityNick)	2
(name, NewCityName)	1
(nick, NewCityNick)	1

Table 2. N_{afo} table (Xtasy model) for the tagged region graph of Fig.3

(root, DB)	(persons, Any)	1
(root, DB)	(cities, Any)	1
(persons, Any)	(person, Person)	1
(person, Person)	(name, PersonName)	1
(name, PersonName)	(fullName, Any)	1
(name, PersonName)	(gensName, Any)	1
(cities, Any)	(city, City)	3
(city, City)	(name, OldCityName)	.66
(city, City)	(nick, OldCityNick)	.66
(city, City)	(name, NewCityName)	.33
(city, City)	(nick, NewCityNick)	.33
(name, OldCityName)	(ancientName, Any)	1
(name, OldCityName)	(modernName, Any)	1

3.3 Match Occurrences, ECLSs, and ETLSs

Estimation functions estimate data distribution in query result by means of sequences of match occurrences, which are formally defined as follows.

Definition 4. *A match occurrence o is a pair $((l, r), m)$, where (l, r) is a tagged region, and m is the multiplicity of the occurrence; a match occurrence $o = ((l, r), m)$, then, says that m nodes labeled l and belonging to region r are part of the result*[3]*.*

Estimation functions manipulate sequences of match occurrences, which are called ECLSs, and can be further organized in more complex structures called ETLSs.

Definition 5. *An ECLS ecls is a list of match occurrences $ecls = \{o_1, \ldots, o_n\}$, such that $\nexists i, j \in 1, \ldots, n \mid o_i = (l_i, r_i, m_i), o_j = (l_i, r_i, m_j), i \neq j$.*

Definition 6. *An ETLS (Extended Tuple Label Sequence) is a list of pairs $(\$v, \{e\})$, where $\$v$ is a distinct variable symbol, and $\{e\}$ is a list of ECLSs.*

ETLSs collect estimations about all tuples produced by the system. Two key points must be noted: first, each variable is bound to a sequence (possibly, a singleton) of ECLSs, in order to support the cardinality estimation of groups; second, the ECLS associated with a variable contains all the match occurrences found for the variable, hence only one ETLS is generated during query cardinality estimation.

The following example shows sample ECLSs and ETLSs.

[3] For the sake of simplicity, match occurrences will be denoted in the form (l, r, m) (instead of $((l, r), m)$).

Example 2. Consider the following query clause:

```
for $c in input()//city,
    $n in $c/name
```

The estimation function first estimates the result for the path expression `input()//city`, hence returning the following ECLS:

$$\{(city, City, 3)\}\ .$$

This ECLS is then bound to the $\$c$ variable symbol, hence generating the following ETLS:

$$\{(\$c : \{\{(city, City, 3)\}\})\}\ .$$

The estimation function, then, scans the match occurrences bound to $\$c$ to find those whose tagged regions in the graph have name children. The multiplicity of these new occurrences is computed by using the statistical information taken from the graph (e.g., N_{afo} and N_{el} in the case of the model of Xtasy). The resulting ETLS is the following:

$$\begin{aligned}\{(\$c : &\quad \{\{(city, City, 3)\}\}),\\ (\$n : &\quad \{\{(name, OldCityName, 2),\\ &\quad (name, NewCityName, 1)\}\}\}\ .\end{aligned}$$

3.4 Cardinality Notions

One key point in any estimation model is what cardinality notion is used, i.e., how the size measures returned by the model should be interpreted. The most common operational model for XML queries (at least, for queries involving variables) is based on the construction of tuples carrying the values bound to variables [1]; since usual optimization heuristics are based on the minimization of the number of granules generated during query evaluation, the *natural* cardinality notion is the number of such granules (e.g., [5]).

The proposed framework embodies the vision of intermediate tuple generation, hence it supports the number of generated tuples as cardinality notion. The way this number is computed from ETLSs depends on the specific model being considered. However, the framework contains a general purpose cardinality function $\|\cdot\|$.

The following example shows how tuple cardinality can be computed from ETLSs.

Example 3. Consider the query of the previous example, and the resulting ETLS, which is reported below.

$$\begin{aligned}\{(\$c : &\quad \{\{(city, City, 3)\}\}),\\ (\$n : &\quad \{\{(name, OldCityName, 2),\\ &\quad (name, NewCityName, 1)\}\}\}\ .\end{aligned}$$

This ETLS estimates the distribution of data into bound variables; variables are organized in twigs, which may eventually be simple paths (as in this case). The number of generated tuples can be estimated as the number of distinct twig instances, which is computed by multiplying the multiplicity of match occurrences bound to leaf variables of the same twig. In this case, the tuple cardinality is the cardinality of the variable $\$n$, hence the framework correctly predicts that three tuples will be generated.

3.5 Correlation

The correlation problem refers to the need of correlating estimations coming from distinct branches of the same twig. Consider, for example, the following query clause:

```
for $x in input()/a,
    $y in $x/b,
    $z in $y/c,
    $w in $y/d
```

This clause matches a two-branch twig against an hypothetical document; in order to correctly predict the number of tuples in the result it is necessary to correlate the estimation for the branch b/c with the estimation for the branch b/d. Without such a correlation, computing the number of tuples represented by a given ETLS would require the model to multiply the multiplicity of $\$z$ with that of $\$w$ (cross product hypothesis), hence introducing many potential errors.

Twig branch correlation can be performed by using regions. The idea is the following. Once estimated the cardinality of the twig branches, the number of generated tuples can be obtained from the resulting ETLS by identifying the variables having a common root variable, and multiplying the multiplicity of those match occurrences sharing the same parent region.

Example 4. Consider the following query fragment:

```
for $c in input()//city,
    $n in $c/name,
    $nick in $c/nick,
```

This query fragment retrieves, for each city element in the database, its name and the list of its nicknames. By evaluating this clause on the sample document of Figure 1, the framework produces the following ETLS:

$$\begin{aligned}\{\$c : \{\{ & (city, City, 3)\}\},\\ \$n : \{\{ & (name, OldCityName, 2)\},\\ & (name, NewCityName, 1)\}\}\}\\ \$nick : \{\{ & (nick, OldCityNick, 2),\\ & (nick, NewCityNick, 1)\}\}\ .\end{aligned}$$

Without any correlation, the predicted number of tuple would be 6, which is clearly wrong (the right number is 3). By correlating nick elements and name elements on the basis of their parent region, the framework can correctly estimate the number of tuples as 3.

Correlation affects the tuple cardinality computing function, as well as other facilities of the framework: the cardinality of an ETLS is computed by multiplying the multiplicity of correlated match occurrences only, independent variables (e.g., variables bound to different documents) being considered as fully correlated (each match occurrence is correlated to any other).

3.6 Group Cardinality Estimation

Group cardinality estimation refers to the problem of estimating the dimension of sets created by the let binder, and, more generally, by the use of free path expressions outside the binding clauses for and let. In Section 2 three main issues were identified about groups: the estimation of the number of distinct groups; the correlation between each group and the variable instance, which it depends on; and the estimation of the distribution of data into each group.

The number of groups created by the let binder is equal to the multiplicity of the variable on which the groups depend. Consider, for example, the query fragment shown below:

```
for $c in input()//city,
let $n_list := $c/name,
```

For each city node bound to c, a distinct $$n_list$ group is created.

Thus, the framework computes the number of groups by using the following function, which sums multiplicity of match occurrences for a single variable[4]:

$$\|etls(\$c)\| = \begin{cases} \Sigma_{(l,r_l,m_l)\in e} m_l & \text{if } etls(\$c) = \{e\} \text{ and} \\ & \$c \in ungrouped(etls) \\ n & \text{if } etls(\$c) = \{e_1, \ldots, e_n\} \\ & \text{and } \$c \in grouped(etls). \end{cases}$$

The second case in the definition is necessary when the root variable of the let binder is itself the result of the application of another let binder, as in the fragment shown below:

```
for $c in input()//city,
let $n_list := $c/name,
let $notes := $n_list/notes,
```

In this particular case, the number of groups is equal to the number of groups of the root variable ($$n_list$).

[4] *grouped*(*etls*) is the set of variables in *etls* bound by a let clause, while *ungrouped*(*etls*) is the set of variables in *etls* bound by a for clause.

Once computed the number of groups, the framework must create each group and estimate the data distribution inside it. The group creation algorithm is based on the correlation function, and, performs the following steps: the algorithm, first, collects all match occurrences of the let right member into a set $\mathcal{S}$, and creates m empty groups, where m is the estimated number of groups; then, it correlates each occurrence o in $\mathcal{S}$ with the root match occurrences, and distributes them accordingly. The following example illustrates the group creation process.

Example 5. Consider our well-known query fragment:

```
for $c in input()//city,
let $n_list := $c/name,
```

By estimating for clause cardinality on the sample document of Fig. 1, the framework generates the following ETLS:

$$\{\$c : \{\{(city, City, 3)\}\}\} \ .$$

The list of match occurrences collected for the let path expression is the following:

$$\begin{aligned}&\{(name, OldCityName, 2),\\&(name, NewCityName, 1)\} \ .\end{aligned}$$

The framework creates three groups, and distributes match occurrences as shown below.

$$\begin{aligned}\{\$n\_list : \{\{ &(name, OldCityName, 1)\},\\&\{(name, OldCityName, 1)\}\}\\&\{(name, NewCityName, 1)\}\} \ .\end{aligned}$$

3.7 Predicate Selectivity Estimation

The estimation of predicate selectivity for XML queries shows two main issues. The first issue concerns the estimation process itself as well as the nature of selectivity factors. As already discussed, XML documents have an irregular structure, where tags and values are distributed in a way far from being uniform. As a consequence, the uniform distribution hypothesis is not suitable for predicates on XML data. Moreover, queries can contain *value* predicates (e.g., `data($y) > 1982`, where $\$y$ is bound to year elements) and *structural* predicates about the existence of children nodes with a given label (e.g., `book[publisher]`); finally, even if we consider only value predicates, variables can be bound to heterogeneous nodes, for instance $\$a$ bound to author and publisher nodes. Hence, the selectivity factor for a given predicate P should be a function of structural information.

Given that, the framework supports selectivity factors as functions of the label and the region of a given match occurrence. This choice allows the framework to take structural information (even type information if an intensional partitioning is used) into account, hence increasing the accuracy of the cardinality estimation. The following example clarifies this point.

Example 6. Consider the following query fragment:

```
for $c in input()//city,
    $n in op:union($c//ancientName,
                   $c//modernName)
where data($n) = "Monticello"
```

The predicate `data($n) = "Monticello"` applies to ancientName and modernName elements, and its selectivity factor depends on the tag of the subject node, since values can have different distributions in ancientName and modernName elements (e.g., "Monticello" is a quite common name for small Italian villages, even though their Latin or medieval names were slightly different).

Selectivity factors for unary predicates can be defined as follows.

Definition 7. *Given a unary predicate P, the selectivity factor of P $psf[P]$ is a function*

$$psf[P] : label \times region \to [0,1]$$

that, given a label l and a region r, returns a real number belonging to [0,1].

This definition naturally leads to histogram-based selectivity factors, even though the model designer is free to choose the preferred way of collecting and storing statistics. Histograms can be built and managed by using well-known techniques, without the need for particular changes.

The following example illustrates unary predicate selectivity factors.

Example 7. Consider the query fragment of the previous example, where variable $\$n$ is bound to ancientName as well as modernName elements. Assuming that statistics are gathered from a bigger document than that of Fig. 1, the selectivity factor for the predicate $P \equiv data(\$n) = "Monticello"$ could be the following:

$$psf[P] = \begin{cases} (ancientName, r_1) \to 0.2 \\ \dots \\ (ancientName, r_5) \to 0.07 \\ (modernName, r_6) \to 0.75 \\ \dots \\ (modernName, r_9) \to 0.67\ . \end{cases}$$

The second main issue about predicate selectivity estimation concerns the way these factors are applied to ETLSs in the framework. For instance, given the predicate `data($n) = "Monticello"`, the values returned by $psf[P]$ are used for decreasing the multiplicity of match occurrences bound to $\$n$. Still, this is not sufficient, as the predicate cuts the number of twig instances collected on data; hence, the multiplicity decrease should be applied also to any directly or indirectly dependent variable ($\$c$ only in the example). For applying this decrease and preserving accuracy, multiplicity decrease propagation should be based on the correlation mechanism previously described.

As a consequence, the framework offers a predicate selectivity factor application function, which, starting from the predicate variable, scans match occurrences, applies the right factor, and reapplies the transformation to directly or indirectly dependent variables. The following example shows how selectivity factors are applied.

Example 8. Consider again the query fragment of Example 6, and the selectivity factor of Example 7; assume that the for clause estimation returns the following ETLS:

$$\begin{aligned}\{\$c : \{\{\ & (city, r_1, 2), (city, r_3, 1), (city, r_4, 1)\}\},\\ \$n : \{\{\ & (ancientName, r_5, 2), (modernName, r_6, 1),\\ & (modernName, r_9, 1)\}\}\}\ .\end{aligned}$$

By applying the selectivity factor of Example 7, the framework generates the following ETLS:

$$\begin{aligned}\{\$c : \{\{\ & (city, r_1, .14), (city, r_3, .75), (city, r_4, .67)\}\},\\ \$n : \{\{\ & (ancientName, r_5, .14), (modernName, r_6, .75),\\ & (modernName, r_9, .67)\}\}\}\ .\end{aligned}$$

(we assume that r_5 correlates to r_1, and that r_6 and r_9 correlate to r_3 and r_4 respectively).

The following example concludes the description of the estimation approach: the next Section will present some experimental results about the statistical model of Xtasy.

Example 9. Consider the following query.

```
for $c in input()//city,
    $n in op:union($c//ancientName,
                   $c//modernName),
    $nick in $c/nick
where data($n) != "Roma"
return <otherNick> $nick </otherNick>
```

This query returns the nicknames of cities whose ancient or modern name is different from "Roma" (we assume that there exists a proper selectivity factor).

The size estimator first collects matches for the for clause of the query, hence returning the following ETLS:

$$\begin{aligned}\{\$c : \{\{\ & (city, City, 3)\}\},\\ \$n : \{\{\ & (ancientName, Any, 2)\},\\ & (modernName, Any, 2)\}\}\}\\ \$nick : \{\{\ & (nick, OldCityNick, 2),\\ & (nick, NewCityNick, 1)\}\}\ .\end{aligned}$$

The size estimator, then, applies the selectivity factor, corresponding to the predicate `data($n) != "Roma"`, to this ETLS: the algorithm cuts $\$n$ occurrences by an half, and then tries to propagates the selectivity factor to directly or indirectly dependent variables; since only occurrences in $(nick, OldCityNick)$ correlate with occurrences in $(ancientName, Any)$ or in $(modernName, Any)$, occurrences in $(nick, NewCityNick)$ are cut off from the resulting ETLS, which is the following:

$$\begin{array}{ll}\{\$c : \{\{ & (city, City, 1.5)\}\}, \\ \$n : \{\{ & (ancientName, Any, 1)\}, \\ & (modernName, Any, 1)\}\}\} \\ \$nick : \{\{ & (nick, OldCityNick, 1)\}\}\} \ .\end{array}$$

By applying the tuple cardinality computing function $\| \cdot \|$, the framework estimates the cardinality of this query as $1 * (1 + 1) = 2$, instead of 1: the overestimation error comes from the inaccuracy generated by intensional regions.

4 Experimental Results

This Section presents some experimental results concerning the estimation model of Xtasy, which is an instance of the framework being described in this paper. The statistical model of Xtasy is based on an extensional partitioning of XML data: a region $\mathcal{R} = (h, p)$ is defined as the set of nodes at the level h in the tree, and whose position in the document order is comprised between p and $p + \delta_p$, where δ_p is a tunable parameter.

The experimental results were obtained by running a set of benchmark queries over the XMark Standard dataset [7], and by comparing predictions with the actual result size.

4.1 Benchmark Queries

The experiments performed to validate the prediction model are based on a set of benchmark queries that, unlike usual XML database benchmarks (see [7] for instance), were designed with the purpose of testing the accuracy of size predictions.

The benchmark contains six sets of queries, each of them related to different size prediction issues: **path queries**, which evaluate linear path expressions, both fully specified and with wildcards and closure operators; **twig queries**, which evaluate twig expressions, both fully specified and with closure operators, but without the grouping binder; **twig queries with groups**, which evaluate complex twig queries with the grouping binder (`let ... :=`); **queries with predicates**, which apply unary predicates to the result of twig expressions; **nested queries**; and, **negative queries**, which are used for estimating the accuracy of size predictions on empty queries.

The first five sets contain *positive* queries, i.e., queries having non-empty result, while the last class is formed by *negative* queries, i.e., queries **with** empty result; this design choice, inspired by [3], is motivated by the will to test the accuracy of the model even in the worst conditions.

4.2 Error Metrics

We use two error metrics for evaluating the accuracy of the size model: the *relative* error for positive queries, and the *absolute* error for negative queries, as shown below[5].

$$RE(Q) = \frac{ES(Q) - AS(Q)}{AS(Q)} \ . \tag{1}$$

$$AE(Q) = ES(Q) - AS(Q) \ . \tag{2}$$

4.3 Experimental Results

We perform the benchmark queries on two distinct statistical configurations: we use $\delta_p = 100000$ and $\delta_p = 500000$. For predicate queries, we estimate the accuracy of the size model with and without proper histograms.

Graphs for the six classes of benchmark queries are shown in Fig. 4: in these graphs, relative errors are shown in the form of percentage relative errors.

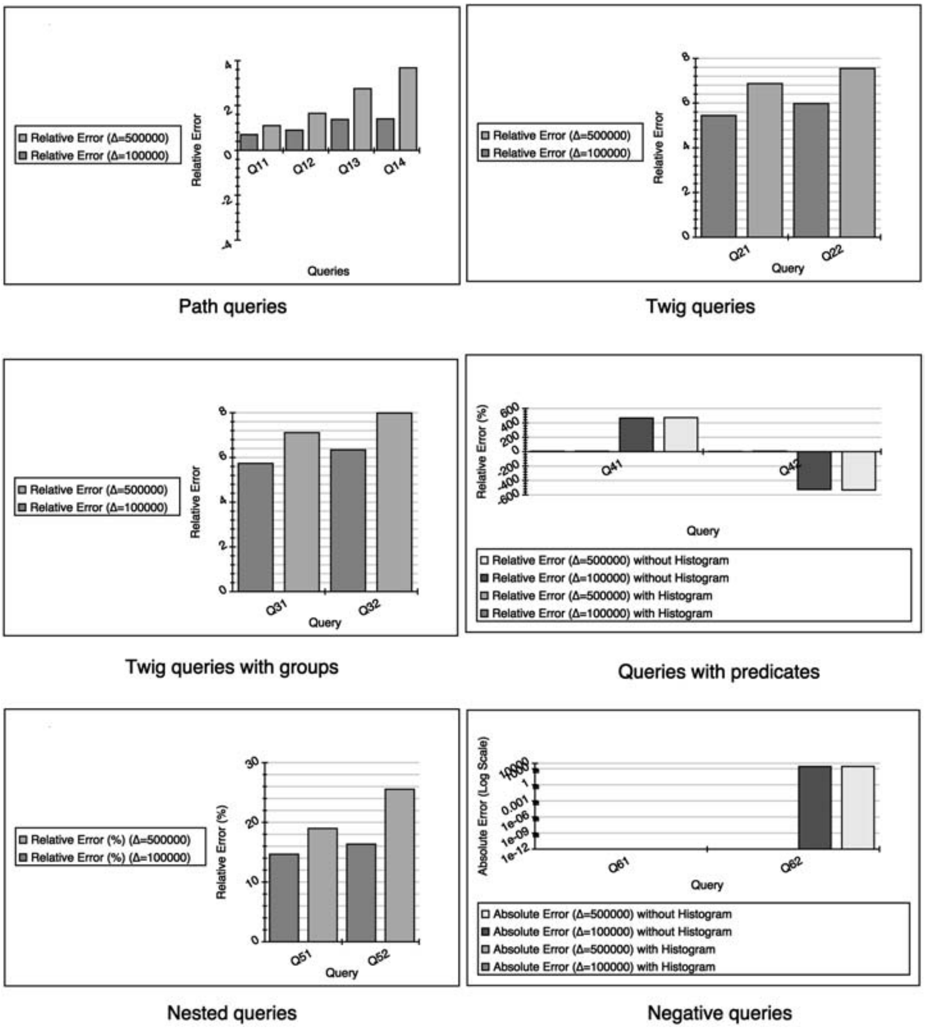
Path Queries. The size estimator virtually introduces no error for queries without //; for queries with //, instead, the system produces some minor errors in the most compressed statistical configurations.

Twig Queries. Unlike path queries, twig queries are more prone to estimation errors, due to the need to perform correlation checks: in particular, when regions contain many nodes, the system overestimates the number of correlated regions, hence leading to a significant estimation error (7.578%).

Twig Queries with Groups. While the previous tests show a *uniform* statistical behavior of the model, i.e., the model tends to overestimate query result size, this query workload shows that the distribution of match occurrences into groups may lead to unexpected underestimation and overestimation of the size of single, which in turn may lead to query cardinality errors. Underestimation errors for groups are not visible in the diagrams, since they are balanced by overestimation errors of other groups, hence they do not appear in aggregate error metrics.

Queries with Predicates. For each dataset, we tested these queries with or without a proper histogram. When an histogram is available, the system virtually introduces no further error in the estimation of `where` clauses; when no histogram is available, instead, the system generates huge estimation errors. These errors are determined by the use of magic numbers (i.e., 0.1 for equality predicates, and 0.33 for other predicates), which are not adequate in the XML setting.

[5] $ES(Q)$ denotes the estimated size of Q, while $AS(Q)$ denotes its actual size.



Path queries

Twig queries

Twig queries with groups

Queries with predicates

Nested queries

Negative queries

Fig. 4. Accuracy tests for benchmark queries

Nested Queries. Due to the complexity of these queries, we test them only with proper histograms for predicates. Experiments show that, while still accurate, the process for synthesizing new statistics can propagate estimation errors, in particular those errors related to the evaluation of twigs and *value-based* joins on heavy compressed statistical configurations.

Negative Queries. This class consists of a *structurally* empty query, and of a query with a false predicate. In the case of the structurally empty query, the system correctly predicts a zero result size; in the case of emptiness induced by the where clause, instead, the absence of a proper histogram leads to an intolerably high error.

5 Related Works

Correlated Subpath Trees. In [3] authors deal with the problem of estimating the number of matches of a twig query over tree structured data. Data trees and twig queries are represented in the same way as node-labeled trees, where leaf nodes are labeled with strings in Σ^* (Σ is an alphabet), while internal nodes are instead labeled with strings in $\Pi \subset \Sigma^*$; as a consequence, queries with closure operators or wildcards (e.g., // and $*$) are not supported by the proposed models.

The main idea behind the paper is the extension of summarization and prediction techniques for substring selectivity [4] to the context of twig queries. Authors first define a summary structure, the *Correlated Subpath Tree*, hosting the most frequent subpaths of the data tree; any subpath in the CST is endowed with its frequency as well as with the hash signature of the set of nodes, which the path is rooted in. Hash signatures are used for correlating subpaths, hence for increasing the accuracy of the prediction. The CST can be pruned by defining a threshold for subpath frequency, and by discarding low frequency paths.

By leveraging on this statistic structure, authors propose four estimation models: pure MO (Maximal Overlap), MOSH (Maximal Overlap with Set Hashing), PMOSH (Piecewise MOSH), and MSH (Maximal Set Hashing). The proposed algorithms have a common structure, and are organized in three phases: path parsing, where a twig query Q is matched over the CST T' to produce a set of paths S; twiglet decomposition, where paths in S are combined to form a set of twiglets (i.e., very small twigs) S', and MO conditioning, where twiglets are combined to form the original query Q. During the first step, path frequencies are retrieved from the CST, while during twiglet decomposition and MO conditioning twiglet and twig frequencies are computed with probabilistic formulae obeying the inclusion-exclusion principle.

The four proposed algorithms differ in the way the three phases are performed. For what concerns path parsing, pure MO, MOSH, and MSH algorithms match root-to-leaf paths in Q with the longest possible subpaths in T', while PMOSH first decomposes Q into segments, and then matches them against T.

For twiglet decomposition, pure MO algorithm just forms twiglets consisting of single paths, while MOSH and PMOSH algorithms form twiglets by combining paths in S via the set hash signature; MSH distinguishes from MOSH and PMOSH because twiglets are formed from paths in Q and from their suffixes in T'.

MO conditioning is the only step where the four proposed algorithms behave exactly the same way.

Differences in the path parsing and the twiglet decomposition steps mainly affect the shape of the resulting twiglets: pure MO twiglets are just paths, as in Niagara's models [2] ; MOSH twiglets are deep but often skinny, while PMOSH twiglets are bushy but often shallow; MSH twiglets, finally, are the result of the trade-off between deepness and bushiness.

Authors experiment the proposed algorithms on two datasets by varying the space reserved for statistics, the database size, and the query workload: three query workloads were considered, consisting respectively of trivial *positive* (e.g., non-empty) queries, non-trivial positive queries, and non-trivial *negative* (e.g., empty) queries. In any test, MOSH and MSH algorithms outperform the others in accuracy.

The main contribution of this paper is the identification of the need to explicitly store correlation information in XML statistics for obtaining good size predictions. On the other hand, the proposed models, while very accurate, deal with a very limited class of twig queries, and the way they can be extended to more general twig queries and to wider query sets appears unclear.

TIMBER Result Size Model. In [8] authors propose two models for estimating the number of matches of a twig query Q over a XML tree T. The proposed models rely on *position histograms* for predicates in a set $\mathcal{P}$. Position histograms can be used for estimating the raw cardinality of simple ancestor/descendant queries. The first model exploits position histograms only, while the second one also uses *coverage histograms*, a kind of structural information for increasing the accuracy of the prediction; unlike the first model, however, this one can be used only when the schema information is available, and, in particular, when the ancestor predicate in a pattern (P_1, P_2) satisfies the *no-overlap* property.

The model based on coverage histograms is much more accurate than the model based only on position histograms; unfortunately, its applicability is limited, and, in particular, it cannot be exploited in recursive documents, where the two proposed models behave badly. Moreover, the estimations are limited to ancestor/descendant paths, and it is not clear how they can be extended to complex twigs involving also parent/child relationships. Finally, the model only deals with twig matching, hence ignoring critical issues such as iterators, binders, nested queries, etc.

Niagara's Models. In [2] authors present the path expression selectivity estimation models employed in Niagara. The models can be used to compute the selectivity of path expressions of the form `a/b/.../f`, i.e., XPath patterns without closure operators (//) and inline conditions; moreover, the models cannot be applied to twigs.

The first model is based on a structure called *path tree*. Since a path tree may have the same size as the database (e.g., when paths in the database are distinct from each other), summarization techniques should be applied to constrain the size of the path tree to the available main memory.

The second model is based on a more sophisticated statistic structure called *Markov table*. This table, implemented as an ordinary hash table, contains any distinct path of length up to m ($m \geq 2$), and its selectivity. As for path trees, the size of a Markov table may exceed the total amount of available main memory, hence summarization techniques are required.

The proposed approaches are quite simple and effective: the Markov table technique, in particular, delivers an high level of accuracy (much more than the pruned suffix tree methods). Unfortunately, they are limited to simple path expressions, and there is no clear way to extend them to twigs or predicates.

StatiX Statistic Model. In [5] authors describe a methodology for collecting statistics about XML documents; the proposed approach is applied in LegoDB for providing statistics about XML-to-relational storage policies, and, to a less extent, in the Galax system for predicting XML query result size.

The StatiX approach aims to build statistics capturing both the irregular structure of XML documents and the *non-uniform* distribution of tags and values within documents. To this purpose, it relies on the schema associated to each document. Indeed, given a XML Schema description $\mathcal{S}$ describing a XML document $\mathcal{T}$, StatiX builds $O(m+n)$ histograms, where m and n are respectively the number of edges and nodes in the graph representation of $\mathcal{S}$. StatiX histograms fall into two categories: *structural* histograms, which describe the distribution and the correlation of non-terminal type instances, and *value* histograms, which, as in the relational case, represent value distribution of simple elements (i.e., elements whose content is a base value).

The StatiX system can tune statistics granularity by applying *conservative* schema transformations to the original XML Schema description, i.e., transformations preserving the class of described documents, and not introducing ambiguity.

6 Conclusions

This paper has described a framework for estimating the cardinality of XML queries. The proposed framework offers tools and algorithms for predicting not only the raw size of query results, but also the distribution of data inside them, hence making the prediction of the size of subsequent operations more accurate. The facilities offered by the framework range from group cardinality estimation to twig branch correlation, and selectivity factor application; by relying on these facilities, the model designer can focus on the definition of accurate and concise statistic summaries.

Acknowledgments

The author would like to thank Dan Suciu for his help during the revision of the paper.

References

1. Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufman, 1999.

2. Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 591–600. Morgan Kaufmann, 2001.
3. Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting twig matches in a tree. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 595–604. IEEE Computer Society, 2001.
4. Paolo Ferragina, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Two-dimensional substring indexing. In *Proceedings of the Twenteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*, 2001.
5. Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jérôme Siméon. Statix: Making XML count. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002, USA*. ACM Press, 2002.
6. Carlo Sartiani. Efficient Management of Semistructured XML Data, 2003. Manuscript draft.
7. Albrecht Schmidt, Florian Waas, Martin Kersten, Daniela Florescu, Ioana Manolescu, Michael J. Carey, and Ralph Busse. The XML Benchmark Project. Technical report, Centrum voor Wiskunde en Informatica, April 2001.
8. Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Proceedings of the 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, 2002*, volume 2287 of *Lecture Notes in Computer Science*, pages 590–608. Springer, 2002.

Author Index